VISIT…

Let Run your Neurons

team LRN

TEXTURING & MODELING

A Procedural Approach

THIRD EDITION

David S. Ebert
F. Kenton Musgrave
Darwyn Peachey
Ken Perlin
Steven Worley

TEXTURING & MODELING

A Procedural Approach

THIRD EDITION

Team LRN

The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling

Series Editor: Brian A. Barsky, University of California, Berkeley

Texturing & Modeling: A Procedural Approach, Third Edition
David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley

Geometric Tools for Computer Graphics
Philip Schneider and David Eberly

Understanding Virtual Reality: Interface, Application, and Design
William Sherman and Alan Craig

Jim Blinn's Corner: Notation, Notation, Notation
Jim Blinn

Level of Detail for 3D Graphics: Application and Theory
David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner

Digital Video and HDTV Algorithms and Interfaces
Charles Poynton

Pyramid Algorithms: A Dynamic Programming Approach to Curves and Surfaces for Geometric Modeling
Ron Goldman

Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation
Thomas Strothotte and Stefan Schlechtweg

Curves and Surfaces for CAGD: A Practical Guide, Fifth Edition
Gerald Farin

Subdivision Methods for Geometric Design: A Constructive Approach
Joe Warren and Henrik Weimer

Computer Animation: Algorithms and Techniques
Rick Parent

The Computer Animator's Technical Handbook
Lynn Pocock and Judson Rosebush

Advanced RenderMan: Creating CGI for Motion Pictures
Anthony A. Apodaca and Larry Gritz

Curves and Surfaces in Geometric Modeling: Theory and Algorithms
Jean Gallier

Andrew Glassner's Notebook: Recreational Computer Graphics
Andrew S. Glassner

Warping and Morphing of Graphical Objects
Jonas Gomes, Lucia Darsa, Bruno Costa, and Luiz Velho

Jim Blinn's Corner: Dirty Pixels
Jim Blinn

Rendering with Radiance: The Art and Science of Lighting Visualization
Greg Ward Larson and Rob Shakespeare

Introduction to Implicit Surfaces
Edited by Jules Bloomenthal

Jim Blinn's Corner: A Trip Down the Graphics Pipeline
Jim Blinn

Interactive Curves and Surfaces: A Multimedia Tutorial on CAGD
Alyn Rockwood and Peter Chambers

Wavelets for Computer Graphics: Theory and Applications
Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin

Principles of Digital Image Synthesis
Andrew S. Glassner

Radiosity & Global Illumination
François X. Sillion and Claude Puech

Knotty: A B-Spline Visualization Program
Jonathan Yen

User Interface Management Systems: Models and Algorithms
Dan R. Olsen, Jr.

Making Them Move: Mechanics, Control, and Animation of Articulated Figures
Edited by Norman I. Badler, Brian A. Barsky, and David Zeltzer

Geometric and Solid Modeling: An Introduction
Christoph M. Hoffmann

An Introduction to Splines for Use in Computer Graphics and Geometric Modeling
Richard H. Bartels, John C. Beatty, and Brian A. Barsky

Team LRN

TEXTURING & MODELING

A Procedural Approach

THIRD EDITION

David S. Ebert
Purdue University

F. Kenton Musgrave
Pandromedia, Inc.

Darwyn Peachey
Pixar Animation Studios

Ken Perlin
New York University

Steven Worley
Worley Laboratories

With contributions from
William R. Mark
University of Texas at Austin

John C. Hart
University of Illinois at Urbana–Champaign

MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER SCIENCE

AMSTERDAM BOSTON LONDON NEW YORK
OXFORD PARIS SAN DIEGO SAN FRANCISCO
SINGAPORE SYDNEY TOKYO

Team LRN

Publishing Director	Diane D. Cerra
Publishing Services Manager	Edward Wade
Senior Production Editor	Cheri Palmer
Senior Developmental Editor	Marilyn Alan
Editorial Coordinator	Mona Buehler
Cover/Text Design	Frances Baca Design
Cover Image	© 2002 Armands Auseklis. MojoWorld by Robert Buttery
Technical Illustration/Composition	Technologies 'n' Typography
Copyeditor	Ken DellaPenta
Proofreader	Jennifer McClain
Indexer	Ty Koontz
Printer	The Maple-Vail Book Manufacturing Group

Figure credits: Figures 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.10: Copyright © 1996 Association for Computing Machinery, Inc. Used with permission. Figure 8.13: Copyright © 1984 Pixar Animation Studios. Figure 15.1: Copyright © 1985 Association for Computing Machinery, Inc. Used with permission.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

Morgan Kaufmann Publishers
An imprint of Elsevier Science
340 Pine Street, Sixth Floor
San Francisco, CA 94104–3205
www.mkp.com

© 2003 by Elsevier Science (USA)
All rights reserved.
Printed in the United States of America

07 06 05 04 03 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, or otherwise—without the prior written permission of the publisher.

Library of Congress Control Number: 2002107243
ISBN: 1–55860–848–6

This book is printed on acid-free paper.

Team LRN

THE UBIQUITY OF PROCEDURAL TECHNIQUES IN COMPUTER GRAPHICS xx
OUR OBJECTIVE xxi
DEVELOPMENT OF THIS BOOK xxii
SOURCE CODE xxii
ACKNOWLEDGMENTS xxii
AUTHOR ADDRESSES xxiii

INTRODUCTION 1
PROCEDURAL TECHNIQUES AND COMPUTER GRAPHICS 1
What Is a Procedural Technique? 1
The Power of Procedural Techniques 2
PROCEDURAL TECHNIQUES AND ADVANCED GEOMETRIC MODELING 2
AIM OF THIS BOOK 3
ORGANIZATION 4

BUILDING PROCEDURAL TEXTURES 7
INTRODUCTION 7
Texture 8
Procedural Texture 11
Procedural versus Nonprocedural 12
Implicit and Explicit Procedures 12
Advantages of Procedural Texture 14
Disadvantages of Procedural Texture 14
The RenderMan Shading Language 15
What If You Don t Use RenderMan? 16
PROCEDURAL PATTERN GENERATION 20
Shading Models 20
Pattern Generation 22
Texture Spaces 24
Layering and Composition 25
Steps, Clamps, and Conditionals 27
Periodic Functions 31

Splines and Mappings 34
Example: Brick Texture 39
Bump- Mapped Brick 41
Example: Procedural Star Texture 46
Spectral Synthesis 48
What Now? 51
ALIASING AND HOW TO PREVENT IT 52
Signal Processing 52
Methods of Antialiasing Procedural Textures 56
Determining the Sampling Rate 57
Clamping 59
Analytic Prefiltering 61
Better Filters 62
Integrals and Summed- Area Tables 63
Example: Antialiased Brick Texture 64
Alternative Antialiasing Methods 66
MAKING NOISES 67
Lattice Noises 69
Value Noise 70
Gradient Noise 72
Value- Gradient Noise 77
Lattice Convolution Noise 78
Sparse Convolution Noise 80
Explicit Noise Algorithms 82
Fourier Spectral Synthesis 82
GENERATING IRREGULAR PATTERNS 83
Spectral Synthesis 85
Perturbed Regular Patterns 89
Perturbed Image Textures 90
Random Placement Patterns 91
CONCLUSION 94

REAL-TIME PROGRAMMABLE SHADING 97
INTRODUCTION 97
What Makes Real- Time Shading Different? 98
Why Use a High- Level Programming Language? 100

What You Need to Learn Elsewhere 101
Real- Time Graphics Hardware 102
Object Space Shading versus Screen Space Shading 103
Parallelism 106
Hardware Data Types 108
Resource Limits 109
Memory Bandwidth and Performance Tuning 110
SIMPLE EXAMPLES 111
Vertex and Fragment Code in the Stanford Shading System 111
Two Versions of the Heidrich/ Banks Anisotropic Shader 112
SURFACE AND LIGHT SHADERS 116
THE INTERFACE BETWEEN SHADERS AND APPLICATIONS 118
MORE EXAMPLES 121
Volume- Rendering Shader 121
Noise- Based Procedural Flame 124
STRATEGIES FOR DEVELOPING SHADERS 129
FUTURE GPU HARDWARE AND PROGRAMMING LANGUAGES 130
LITERATURE REVIEW 131
ACKNOWLEDGMENTS 132

CELLULAR TEXTURING 135
THE NEW BASES 136
IMPLEMENTATION STRATEGY 140
Dicing Space 142
Neighbor Testing 144
The Subtle Population Table 145
Extensions and Alternatives 147
SAMPLE CODE 149

ADVANCED ANTIALIASING 157
INDEX ALIASING 158
An Example: Antialiasing Planetary Rings 163
SPOT GEOMETRY 166
SAMPLING AND BUMPING 170
OPTIMIZATION AND VERIFICATION 173
EMERGENCY ALTERNATIVES 175

PRACTICAL METHODS FOR TEXTURE DESIGN 179
INTRODUCTION 179
TOOLBOX FUNCTIONS 179
The Art of Noise 179
Color Mappings 181
Bump- Mapping Methods 183
THE USER INTERFACE 187
Parameter Ranges 188
Color Table Equalization 189
Exploring the Parameter Domain 192
Previews 194
EFFICIENCY 194
TRICKS, PERVERSIONS, AND OTHER FUN TEXTURE ABUSES 195
Volume Rendering with Surface Textures 195
Odd Texture Ideas 196
D Mapping Methods 197
WHERE WE RE GOING 199

PROCEDURAL MODELING OF GASES 203
INTRODUCTION 203
PREVIOUS APPROACHES TO MODELING GASES 204
THE RENDERING SYSTEM 205
Volume- Rendering Algorithm 206

Illumination of Gaseous Phenomena 207
Volumetric Shadowing 207
ALTERNATIVE RENDERING AND MODELING APPROACHES FOR GASES 208
A PROCEDURAL FRAMEWORK: SOLID SPACES 209
Development of Solid Spaces 209
Description of Solid Spaces 210
Mathematical Description of Solid Spaces 210
GEOMETRY OF THE GASES 211
My Noise and Turbulence Functions 211
Basic Gas Shaping 214
CONCLUSION 224

ANIMATING SOLID SPACES 227
ANIMATION PATHS 228
ANIMATING SOLID TEXTURES 229
Marble Forming 229
Marble Moving 232
Animating Solid Textured Transparency 233
ANIMATION OF GASEOUS VOLUMES 235
Helical Path Effects 236
THREE- DIMENSIONAL TABLES 244
Accessing the Table Entries 245
Functional Flow Field Tables 246
Functional Flow Field Functions 246
Combinations of Functions 251
ANIMATING HYPERTEXTURES 254
Volumetric Marble Formation 255
PARTICLE SYSTEMS: ANOTHER PROCEDURAL ANIMATION TECHNIQUE 257
CONCLUSION 261

VOLUMETRIC CLOUD MODELING WITH IMPLICIT FUNCTIONS 263
CLOUD BASICS 263
SURFACE- BASED CLOUD MODELING APPROACHES 267
VOLUMETRIC CLOUD MODELS 268
A VOLUMETRIC CLOUD MODELING SYSTEM 269
VOLUMETRIC CLOUD RENDERING 272
Cumulus Cloud Models 272
Cirrus and Stratus Clouds 275
Cloud Creatures 278
User Specification and Control 278
ANIMATING VOLUMETRIC PROCEDURAL CLOUDS 279
Procedural Animation 279
Implicit Primitive Animation 280
INTERACTIVITY AND CLOUDS 283
Simple Interactive Cloud Models 283
Rendering Clouds in Commercial Packages 284
CONCLUSION 284

ISSUES AND STRATEGIES FOR HARDWARE ACCELERATION OF PROCEDURAL TECHNIQUES 287
INTRODUCTION 287
GENERAL ISSUES 287
COMMON ACCELERATION TECHNIQUES ... 289
EXAMPLE ACCELERATED/ REAL- TIME PROCEDURAL TEXTURES AND MODELS 291
Noise and Turbulence 291
Marble 292
Smoke and Fog 297
Real- Time Clouds and Procedural Detail 298

CONCLUSION 301

PROCEDURAL SYNTHESIS OF GEOMETRY ... 305
THE L- SYSTEM 307
PARADIGMS GOVERNING THE SYNTHESIS OF PROCEDURAL GEOMETRY 312
Data Amplification 312
Lazy Evaluation 314
THE SCENE GRAPH 315
PROCEDURAL GEOMETRIC INSTANCING ... 321
Parameter Passing 321
Accessing World Coordinates 323
Other Functions 327
Comparison with L- Systems 329
Ordering 330
BOUNDING VOLUMES 330
CONCLUSION 332
Procedural Geometric Modeling and the Web 333
Future Work 333
ACKNOWLEDGMENTS 334

NOISE, HYPERTEXTURE, ANTIALIASING, AND GESTURE 337
INTRODUCTION 337
Shape, Solid Texture, and Hypertexture 337
TWO BASIC PARADIGMS 338
Bias, Gain, and So Forth 338
CONSTRUCTING THE NOISE FUNCTION 340
Computing Which Cubical Cel We re In 341
Finding the Pseudorandom Wavelet at Each Vertex of the Cel 341
Wavelet Coefficients 342
To Quickly Index into G in a Nonbiased Way 343
Evaluating the Wavelet Centered at [i, j, k] 344

CONCLUSION 301

PROCEDURAL SYNTHESIS OF GEOMETRY ... 305
THE L- SYSTEM 307
PARADIGMS GOVERNING THE SYNTHESIS OF PROCEDURAL GEOMETRY 312
Data Amplification 312
Lazy Evaluation 314
THE SCENE GRAPH 315
PROCEDURAL GEOMETRIC INSTANCING ... 321
Parameter Passing 321
Accessing World Coordinates 323
Other Functions 327
Comparison with L- Systems 329
Ordering 330
BOUNDING VOLUMES 330
CONCLUSION 332
Procedural Geometric Modeling and the Web 333
Future Work 333
ACKNOWLEDGMENTS 334

NOISE, HYPERTEXTURE, ANTIALIASING, AND GESTURE 337
INTRODUCTION 337
Shape, Solid Texture, and Hypertexture 337
TWO BASIC PARADIGMS 338
Bias, Gain, and So Forth 338
CONSTRUCTING THE NOISE FUNCTION 340
Computing Which Cubical Cel We re In 341
Finding the Pseudorandom Wavelet at Each Vertex of the Cel 341
Wavelet Coefficients 342
To Quickly Index into G in a Nonbiased Way 343
Evaluating the Wavelet Centered at [i, j, k] 344

RECENT IMPROVEMENTS TO THE NOISE FUNCTION 347
RAYMARCHING 348
System Code: The Raymarcher 349
Application Code: User- Defined Functions 350
INTERACTION 352
Levels of Editing: Changing Algorithms to Tweaking Knobs 352
z- Slicing 353
SOME SIMPLE SHAPES TO PLAY WITH 353
Sphere 353
Egg 354
EXAMPLES OF HYPERTEXTURE 354
Explosions 354
Life- Forms 355
Space- Filling Fractals 357
Woven Cloth 357
ARCHITEXTURE 359
The NYU Torch 362
Smoke 364
Time Dependency 365
Smoke Rings 365
Optimization 367
TURBULENCE 368
ANTIALIASED RENDERING OF PROCEDURAL TEXTURES 369
Background 370
The Basic Idea 370
More Detailed Description 372
The High- Contrast Filter 373
Examples 374
To Sum Up 376
SURFLETS 376
Introduction to Surflets 377
Surflets as Wavelets 378
Finding Visible Surfaces 379

Selective Surface Refinement 380
A Surflet Generator 381
Constructing a Surflet Hierarchy 381
Self- Shadowing with Penumbra 382
Discussion 383
Conclusion 384
FLOW NOISE 384
Rotating Gradients 385
Pseudoadvection 386
Results 387
PROCEDURAL SHAPE SYNTHESIS 387
TEXTURAL LIMB ANIMATION 387
Introduction to Textural Limb Motion 391
Road Map 391
Related Work 392
Basic Notions 392
Stochastic Control of Gesture 392
The System 393
EXAMPLES 395
TEXTURE FOR FACIAL MOVEMENT 395
Background 396
Related Work 397
The Movement Model 398
Movement Layering 401
The Bottom- Level Movement Vocabulary 402
Painting with Actions 403
Using 405
in Movement 405
Same Action in Different Abstractions 408
What Next? 409
CONCLUSION 410

REAL-TIME PROCEDURAL SOLID TEXTURING 413
A REAL- TIME PROCEDURAL SOLID TEXTURING ALGORITHM 413

CREATING AN ATLAS FOR PROCEDURAL SOLID TEXTURING 416
AVOIDING SEAM ARTIFACTS 419
IMPLEMENTING REAL- TIME TEXTURING PROCEDURES 421
APPLICATIONS 425
ACKNOWLEDGMENTS 427

A BRIEF INTRODUCTION TO FRACTALS 429
WHAT IS A FRACTAL? 430
WHAT ARE FRACTALS GOOD FOR? 434
FRACTALS AND PROCEDURALISM 436
PROCEDURAL fBm 436
MULTIFRACTAL FUNCTIONS 438
FRACTALS AND ONTOGENETIC MODELING 442
CONCLUSION 444

FRACTAL SOLID TEXTURES: SOME EXAMPLES 447
CLOUDS 448
Puffy Clouds 448
A Variety of fBm 450
Distortion for Cirrus Clouds and Global Circulation 453
The Coriolis Effect 458
FIRE 460
WATER 461
Noise Ripples 461
Wind- Blown Waters 463
EARTH 466
Sedimentary Rock Strata 466
Gaea: Building an Entire Planet 467
Selene 473

RANDOM COLORING METHODS 477
Random fBm Coloring 477
The GIT Texturing System 478
An Impressionistic Image Processing Filter 479
The Multicolor Texture 483
PLANETARY RINGS 485

PROCEDURAL FRACTAL TERRAINS 489
ADVANTAGES OF POINT EVALUATION 489
THE HEIGHT FIELD 491
HOMOGENEOUS fBm TERRAIN MODELS 495
Fractal Dimension 495
Visual Effects of the Basis Function 497
HETEROGENEOUS TERRAIN MODELS 498
Statistics by Altitude 500
A Hybrid Multifractal 502
Multiplicative Multifractal Terrains 505
CONCLUSION 506

QAEB RENDERING FOR PROCEDURAL MODELS 509
INTRODUCTION 509
QAEB TRACING 510
PROBLEM STATEMENT 511
PRIOR ART 512
THE QAEB ALGORITHM 513
ERROR IN THE ALGORITHM 513
NEAR AND FAR CLIPPING PLANES 514
CALCULATING THE INTERSECTION POINT AND SURFACE NORMAL 515
ANTIALIASING 516
A SPEEDUP SCHEME FOR HEIGHT FIELDS 516

SHADOWS, REFLECTION, AND REFRACTION 517
PERFORMANCE 518
QAEB- TRACED HYPERTEXTURES 520
Clouds 521
BILLOWING CLOUDS, PYROCLASTIC FLOWS, AND FIREBALLS 523
Fireballs 525
Psychedelic Clouds 525
CONCLUSION 525

ATMOSPHERIC MODELS 529
INTRODUCTION 529
BEER S LAW AND HOMOGENEOUS FOG 531
EXPONENTIAL MIST 532
A RADIALLY SYMMETRIC PLANETARY ATMOSPHERE 534
A MINIMAL RAYLEIGH SCATTERING APPROXIMATION 536
TRAPEZOIDAL QUADRATURE OF 539
GADD AND RENDERMAN 539
IMPLEMENTATION 539
NUMERICAL QUADRATURE WITH BOUNDED ERROR FOR GENERAL RADIAL GADDs 542
CONCLUSION 544

GENETIC TEXTURES 547
INTRODUCTION: THE PROBLEM OF PARAMETER PROLIFERATION 547
A USEFUL MODEL: AESTHETIC 548
SPACES 548
CONTROL VERSUS AUTOMATICITY 549

A MODEL FROM BIOLOGY: GENETICS AND EVOLUTION 550
The Analogy: Genetic Programming 552
Implementation 555
INTERPRETATION OF THE ROOT NODE 555
THE LIBRARY OF GENETIC BASES 556
OTHER EXAMPLES OF GENETIC PROGRAMMING AND GENETIC ART 557
A FINAL DISTINCTION: GENETIC PROGRAMMING VERSUS GENETIC ALGORITHMS 558
CONCLUSION 560

MOJOWORLD: BUILDING PROCEDURAL PLANETS 565
INTRODUCTION 565
FRACTALS AND VISUAL COMPLEXITY 567
Building Mountains 567
Building Planets 568
Building a Virtual Universe 571
WHAT IS A FRACTAL? 571
Self- Similarity 572
Dilation Symmetry 573
Random Fractals 574
A BIT OF HISTORY OF FRACTAL TERRAINS 575
The Mathematics 575
Mathematical Imaging of Fractal Terrains 576
The Computer Graphics Research Community 576
The Literature 579
The Software 579
Disclaimers and Apologies 580
The Present and Future 581
BUILDING RANDOM FRACTALS 582
The Basis Function 583

Fractal Dimension: Roughness 583
Octaves: Limits to Detail 584
Lacunarity: The Gap between Successive Frequencies 585
ADVANCED TOPICS 587
Dimensions: Domain and Range 587
Hyperspace 588
The Basis Functions 590
The Seed Tables 597
Monofractals 599
Multifractals 600
Function Fractals 602
Domain Distortion 604
Distorted Fractal Functions 606
Crossover Scales 606
Driving Function Parameters with Functions 607
USING FRACTALS 608
Textures 608
Terrains 609
Displacement Maps 610
Clouds 611
Planets 611
Nebulae 611
THE EXPRESSIVE VOCABULARY OF RANDOM FRACTALS 611
EXPERIMENT! 613
THE FUTURE 614

ON THE FUTURE: ENGINEERING THE APPEARANCE OF CYBERSPACE 617
INTRODUCTION 617
CLAIMS 618
THE FRACTAL GEOMETRY OF CYBERSPACE 620
CONCLUSION 623

APPENDICES 625

BIBLIOGRAPHY 629
CONTRIBUTORS 687

FOREWORD

What is a realistic image? This is an age-old question in art, and a contemporary question in computer graphics. This book provides a modern answer involving the computer and a new definition of realism.

The classic definition of realism has been veridical realism. Does the picture pass the comparison test? That is, would an observer judge the picture to be real? This is traditionally described by Pliny's story (in Book 35 of his *Natural History*) of the ancient painter Zeuxis who painted a picture of a boy carrying some grapes, and when the birds flew up to the picture, he approached the work and, in irritation, said, "I have painted the grapes better than the boy, for if I had rendered him perfectly, the birds would have been afraid."

Nowadays the ultimate in fooling the eye is special effects in the movies. Almost every movie involves hundreds of special effects that are seamlessly combined with live action. It is impossible to tell what is real and what is synthesized. Equally amazing are full-length, computer-generated pictures such as *Shrek*. Although few would be fooled into believing these worlds are real, it is more the artistic choice of the storyteller than a technological limitation. A major achievement in the last two decades is that computers allowed us to achieve veridical realism of imagined scenes.

Besides direct comparison, there are other definitions of real. Masters such as Vermeer used optical devices to aid them in painting realistic pictures, and modern photorealists such as Richard Estes paint over a projected image of a photograph. Thus, another definition of real is to be traced or copied from an image. In this sense the montage of composite layers in a movie is photoreal, since different elements come from different film sequences. There are many other definitions of realism. For example, real can mean a choice of subject matter, such as everyday life versus a myth or an idealized form.

The definition of realism that I like the most is the one I first heard from my colleague, then at Pixar, Alvy Ray Smith: he claimed photorealism was roughly equivalent to visual complexity. Two factors underlie visual complexity, diversity in the types of primitives and their sheer numbers. This definition resonates with computer scientists, since computers are very good at both supporting a wide range of

Team LRN

computational primitives and processing enormous amounts of data. This book is about using the computer to generate visual complexity, an approach called procedural modeling.

The causes of visual complexity in the computer-generated image are the ingredients of perception: color, texture, edges, depth, and motion. The equivalents in object-space, or in the scene, are color, pattern, reflection, illumination, shape, and motion. All these factors come together in composite materials such as wood, stone, and cloth and in natural phenomena such as clouds, steam, smoke, fire, water, landscapes, and planetoids. Procedural models for these myriad objects are the subjects of this book.

Why are computers so good at generating visual complexity? The reason is profound as well as practical.

First, computers expand the types of models that may be used. For example, a surface may be defined as the zeros of an implicit function of x, y, and z. The simplest implicit functions are quadratic functions of the coordinates and define the famous quadric surfaces: spheres, cones, cylinders, and so on. Using a modern programming language with all its built-in functions and arithmetic operators, much more complicated expressions are just as easy to form and to evaluate. Perlin's hypertextured surfaces arise from this flexibility and generality.

Second, computers can generate many from few. A few parameters (or a small amount of geometry) magically expand into a large, detailed model. "Data amplification" gives the user tremendous power, leveraging their efforts and offloading tedious specification of every single detail. A related concept is Kolmogorev complexity, or the smallest program capable of generating a given function. A very few lines of code can produce beautiful pictures. The hacker "demoscene" dramatically illustrates this idea. Here programmers are given the constraint that the size of the file containing both code and data (models, textures, sounds) must fit in less than 64KB. From this file emerges a richly detailed animation.

Third, computational models are by necessity discrete and finite. Although at first this may seem like a limitation, since computational procedures must approximate continuous mathematics and physics, it may in fact open up many new possibilities. For example, an approximation of a smooth curve may be generated from an n-sided polygon by a simple corner-cutting algorithm. Each step consists of cutting off all the corners of the polygon, replacing a vertex with an edge and two new vertices. After an infinite number of iterations of the cutting procedure, the input polygon will converge to a smooth curve. However, on a computer, we can never perform an infinite number of steps, so perfectly smooth curves can never be constructed. Once we give up on idealized mathematical smoothness, we can generalize

Team LRN

corner-cutting polygons to subdividing 3D polyhedral meshes; although these new algorithms do not form smooth objects, a whole new universe of different types of curves and surfaces can now be generated on the computer.

For these reasons procedural modeling is a very powerful new tool that is enabled by the computer. This approach is what is fundamentally different about computer graphics and traditional forms of image making.

An important issue that remains, the Achilles heel of this approach, is controllability. Whether it is a physical simulation with its initial or boundary conditions, or a procedural model with its parameters, the end result must serve the needs of the user. The benefit of filling in detail automatically comes at a cost: the user loses control over the details. The need for controllability drives the development of interactive, what-you-see-is-what-you-get systems. This tension between precise control and programmed complexity remains an interesting research issue. In practice, virtual characters are usually modeled manually, and their motion is generated using keyframe animation. However, buildings, landscapes, and crowds are increasingly being generated using procedural techniques.

This new edition is particularly timely. Although the interest in procedural modeling subsided for a while, there has suddenly been an explosion of new research and development. Processing power continues to increase faster than human modeling power, and as a result models produced procedurally have a level of detail that cannot be produced by hand. New approaches have also emerged: machine learning has been coupled with procedural modeling so that it is now possible to analyze and then synthesize textures, shapes, motions, and styles from examples.

Another major new development is programmable graphics hardware. Graphics processing units, or GPUs, have always been increasing in performance much faster than CPUs. In the last few years, GPUs switched from a fixed-functionality to a flexible, programmable graphics pipeline. Now it is possible to download procedural models into these processors. Currently, GPUs are mostly limited to evaluating procedural texture and reflection models, but in the not too distant future they will be able to produce geometry and motion procedurally as well. Procedural models thus have technology on their side, since they use less bandwidth and communication resources than traditional approaches to graphics systems.

This book describes the complete toolbox of procedural techniques from theory to practice. The authors are the key inventors of the technology and some of the most creative individuals I know. This book has always been my favorite computer graphics book, and I hope you will enjoy it as much as I have.

Pat Hanrahan

Team LRN

PREFACE

This book imparts a working knowledge of procedural approaches in texturing, modeling, shading, and animation and demonstrates their use in high-quality offline and real-time applications. These include two-dimensional and solid texturing, hypertextures, volume density functions, and fractals. Readers are provided with the details often omitted from technical papers, enabling them to explore how the procedures are designed to produce realistic imagery. This book also contains many useful procedures and descriptions of how these procedures were developed. Readers will gain not only a powerful toolbox of procedures upon which to build a library of procedural textures and objects, but also a better understanding of how these functions work and how to design them. With procedures like noise[1] and *turbulence* and an understanding of procedural design techniques, readers will be able to design more complex procedures to produce realistic textures, gases, hypertextures, landscapes, and planets. The procedural techniques are explained not by people who have read some technical papers and tried to decipher them, but by the people who develop the techniques, wrote the seminal papers in the area, and have worked with procedural design for more than 10 years.

THE UBIQUITY OF PROCEDURAL TECHNIQUES IN COMPUTER GRAPHICS

Procedural modeling, texturing, and shading are ubiquitous, vital tools for creating realistic graphics and animation in applications ranging from movie special effects to computer games. Procedural techniques were originally introduced to produce textures for objects. With the introduction of three-dimensional texturing techniques (solid texturing) by Ken Perlin, Darwyn Peachey, and Geoffrey Gardner in 1985, the use of procedural techniques exploded. Realistic images containing marble, wood, stone, and clouds were now possible. Procedural techniques became an area of

1. Ken Perlin won a Technical Achievement Award from the Academy of Motion Picture Arts and Sciences in 1997 for the development of his noise function.

Team LRN

active research in computer graphics. Many programmers and researchers developed their own procedures to simulate natural materials and natural phenomena. What was lacking, however, was a clear understanding of the design of procedural techniques and of the primitive stochastic functions that were used to produce these amazing images. Since the mid-1980s, the use of procedural techniques has grown rapidly, and they can now be used to actually define the geometry of objects such as water, fire, gases, planets, and tribbles.

The use of procedural techniques is not limited to still images; they have been used successfully for animation and the simulation of natural phenomena such as fog, fire, water, and atmospheric patterns. The animation of procedural models requires knowledge of both animation principles and the characteristics of the procedural model. Procedural animation is a powerful technique for producing complex, realistic motion.

With the advent of low-cost programmable graphics processors, procedural techniques have become vital to creating high-quality effects in interactive entertainment and computer games. As of late 2002, it is now possible to implement most of the techniques presented in this book as controllable, interactive procedures that can harness the power of programmable PC graphics to run at real-time rates.

OUR OBJECTIVE

The objective of this book is to provide readers with an understanding and working knowledge of procedural techniques in texturing, modeling, and animation. This book describes the current state of procedural techniques and provides readers with the challenge and information necessary to extend the state of the art. Readers will gain the following from the book:

- A thorough understanding of procedural techniques for solid texturing
- An insight into different design approaches used by the authors in designing procedures
- A toolbox of procedures and basic primitive functions (noise, turbulence, etc.) to produce realistic images
- An understanding of several advanced procedural approaches for modeling object geometry (hypertextures, gases, fractals)
- An introduction to animating these procedural objects and textures
- An understanding of how to adapt these techniques to commodity graphics hardware

Team LRN

DEVELOPMENT OF THIS BOOK

At SIGGRAPH '91, Darwyn Peachey and David Ebert first discussed the need for a course to explain how texture and modeling procedures are designed to create impressive images of wood and marble objects, gases, landscapes, and planets. There were some classic papers on these topics, but they were lacking in several respects. First of all, not enough detail was given to enable readers to reproduce the results in most of the papers. Second, if an image could be reproduced, readers still didn't know how to modify the procedure to get a different effect. Finally, the reason why a procedure produced a given image was unclear. There seemed to be some "magic" behind the development of these procedures. From this discussion, our course at SIGGRAPH '92 arose. There was great demand for the course at both SIGGRAPH '92 and SIGGRAPH '93. We have received useful feedback, thanks, and requests for more material on this topic from the attendees of these courses.

With the success of these courses, we decided to produce the first edition of this book. It was similar in nature to our course notes, but greatly expanded and revised. The second edition contained new chapters discussing work from 1994 to 1998.

There were two motivations for the third edition. First, we wanted to expand, update, and, in essence, complete this book to be *the* reference and source for procedural techniques in computer graphics. Second, we wanted to describe the developments that have been made in procedural techniques in the past five years and the applications of these techniques to games and other real-time graphics applications. Two authors, William Mark and John Hart, have been added to more completely cover interactive procedural techniques.

SOURCE CODE

All of the source code for the examples in this book is available on the Web site for the book at *www.mkp.com/tm3*.

ACKNOWLEDGMENTS

We wish to thank the reviewers of our book proposal, who made valuable suggestions to improve this book. We also wish to thank the attendees of our SIGGRAPH courses, and our students, for feedback and suggestions that led to its development. Susan Ebert, Judy Peachey, and Rick Sayre were of immeasurable help in reviewing

Team LRN

parts of this book. Holly Rushmeier helped with figure creation. Larry Gritz provided RenderMan translations for Ken Musgrave's code and coauthored Chapter 18. Nikolai Svakhine and Joe Kniss contributed to Chapter 10. Tom Deering helped Darwyn Peachey with his Illustrator figures, and Benjamin Zhu, Xue Dong Wong, Rajesh Raichoudhury, and Ron Espiritu helped Ken Perlin in the preparation of his chapters. Bill Mark thanks Pat Hanrahan for sharing his knowledge and advice, and for providing him with the opportunity to spend time writing his chapter. We also wish to thank our families, friends, and coworkers for support during the development of this book. Finally, we wish to thank the staff at Morgan Kaufmann for producing this third edition.

David S. Ebert

AUTHOR ADDRESSES

Our current physical and email addresses are given below.

Dr. David S. Ebert
School of Electrical and
Computer Engineering
1285 EE Building
Purdue University
West Lafayette, IN 47906
ebertd@purdue.edu

Dr. F. Kenton Musgrave
FractalWorlds.com
15724 Trapshire Court
Waterford, VA 20197–1002
musgrave@fractalworlds.com

Darwyn Peachey
Pixar
1200 Park Avenue
Emeryville, CA 94608
peachey@pixar.com

Dr. Ken Perlin
NYU Media Research Lab
719 Broadway, Room 1202
New York, NY 10003
perlin@nyu.edu

Steven Worley
Worley Laboratories
405 El Camino Real #121
Menlo Park, CA 94025
spworley@worley.com

Team LRN

1

INTRODUCTION

DAVID S. EBERT

PROCEDURAL TECHNIQUES AND COMPUTER GRAPHICS

Procedural techniques have been used throughout the history of computer graphics. Many early modeling and texturing techniques included procedural definitions of geometry and surface color. From these early beginnings, procedural techniques have exploded into an important, powerful modeling, texturing, and animation paradigm. During the mid- to late 1980s, procedural techniques for creating realistic textures, such as marble, wood, stone, and other natural material, gained widespread use. These techniques were extended to procedural modeling, including models of water, smoke, steam, fire, planets, and even tribbles. The development of the RenderMan shading language (Pixar 1989) greatly expanded the use of procedural techniques. Currently, most commercial rendering and animation systems even provide a procedural interface. Procedural techniques have become an exciting, vital aspect of creating realistic computer-generated images and animations. As the field continues to evolve, the importance and significance of procedural techniques will continue to grow. There have been two recent important developments for real-time procedural techniques: increased CPU power and powerful, *programmable* graphics processors (GPUs), which are available on affordable PCs and game consoles. This has started an age where we can envision and realize interactive complex procedural models and effects.

What Is a Procedural Technique?

Procedural techniques are code segments or algorithms that specify some characteristic of a computer-generated model or effect. For example, a procedural texture for a marble surface does not use a scanned-in image to define the color values. Instead, it uses algorithms and mathematical functions to determine the color.

Team LRN

The Power of Procedural Techniques

One of the most important features of procedural techniques is *abstraction*. In a procedural approach, rather than explicitly specifying and storing all the complex details of a scene or sequence, we abstract them into a function or an algorithm (i.e., a procedure) and evaluate that procedure when and where needed. We gain a storage savings, as the details are no longer explicitly specified but implicit in the procedure, and the time requirements for specification of details are shifted from the programmer to the computer. This allows us to create inherent multiresolution models and textures that we can evaluate to the resolution needed.

We also gain the power of *parametric control,* allowing us to assign to a parameter a meaningful concept (e.g., a number that makes mountains rougher or smoother). Parametric control also provides amplification of the modeler/animator's efforts: a few parameters yield large amounts of detail; Smith (1984) referred to this as *database amplification*. This parametric control unburdens the user from the low-level control and specification of detail. We also gain the serendipity inherent in procedural techniques: we are often pleasantly surprised by the unexpected behaviors of procedures, particularly stochastic procedures.

Procedural models also offer *flexibility.* The designer of the procedures can capture the essence of the object, phenomenon, or motion without being constrained by the complex laws of physics. Procedural techniques allow the inclusion in the model of any desired amount of physical accuracy. The designer may produce a wide range of effects, from accurately simulating natural laws to purely artistic effects.

PROCEDURAL TECHNIQUES AND ADVANCED GEOMETRIC MODELING

Geometric modeling techniques in computer graphics have evolved significantly as the field matures and attempts to portray increasingly complex models and the complexities of nature. Earlier geometric models, such as polygonal models, patches, points, and lines, are insufficient to represent this increased complexity in a manageable and controllable fashion. Higher-level modeling techniques have been developed to provide an abstraction of the model, encode classes of objects, and allow high-level control and specification of the models. Many of these advanced geometric modeling techniques are inherently procedural. Grammar-based models (Smith 1984; Prusinkiewicz and Lindenmayer 1990), including graftals and L-systems, allow the specification of a few parameters to simulate complex models of trees, plants, and other natural objects. These models use formal languages to specify complex growth rules for the natural objects.

Team LRN

Implicit surface models—also called blobby molecules (Blinn 1982b), meta-balls (Nishimura et al. 1985), and soft objects (Wyvill, McPheeters, and Wyvill 1986)—are used in modeling organic shapes, complex manufactured shapes, and "soft" objects that are difficult to animate and describe using more traditional techniques. Implicit surfaces were first introduced into computer graphics by Blinn (1982b) to produce images of electron density clouds. They are surfaces of constant value, *isosurfaces,* created from blending primitives (functions or skeletal elements) represented by implicit equations of the form $F(x, y, z) = 0$. Implicit surfaces are a more concise representation than parametric surfaces and provide greater flexibility in modeling and animating soft objects. For modeling complex shapes, several basic implicit surface primitives are smoothly blended to produce the final shape. The detailed geometric shape of the implicit surface is not specified by the modeler/animator; instead, it is procedurally determined through the evaluation of the implicit functions, providing higher-level specification and animation control of complex models.

Particle systems are procedural in their abstraction of specification of the object and control of its animation (Reeves 1983). A particle system object is represented by a large collection (cloud) of very simple geometric particles that change stochastically over time. Therefore, particle systems do use a large database of geometric primitives to represent natural objects ("fuzzy objects"), but the animation, location, birth, and death of the particles representing the object are controlled algorithmically. As with other procedural modeling techniques, particle systems have the advantage of database amplification, allowing the modeler/animator to specify and control this extremely large cloud of geometric particles with only a few parameters. Particle systems are described in more detail in Chapter 8.

These advanced geometric modeling techniques are not the focus of this book. However, they may be combined with the techniques described in this book to exploit their procedural characteristics and evolve better modeling and animation techniques.

Additionally, some aspects of image synthesis are by nature procedural; that is, they can't practically be evaluated in advance (e.g., view-dependent specular shading and atmospheric effects). Our primary focus is *procedural textures, procedural modeling,* and *procedural animation.*

AIM OF THIS BOOK

This book will give you a working knowledge of several procedural texturing, modeling, and animation techniques, including two-dimensional texturing, solid

Team LRN

texturing, hypertextures, volumetric procedural models, fractal and genetic algorithms, and virtual procedural actors. We provide you with the details of these techniques, which are often omitted from technical papers, including useful and practical guidelines for selecting parameter values.

We provide a toolbox of specific procedures and basic primitive functions (noise, turbulence, etc.) to produce realistic images. An in-depth description of noise functions is presented, accompanied by several implementations and a spectral comparison of these functions. Some of the procedures presented can be used to create realistic images of marble, brick, fire, steam, smoke, water, clouds, stone, and planets.

ORGANIZATION

This book follows a progression in the use of procedural techniques: from procedural texturing, to volumetric procedural objects, and finally to fractals. In each chapter, the concepts are illustrated with a large number of example procedures. These procedures are presented in C code segments or in the RenderMan shading language.

The details of the design of these procedures are also explained to aid you in gaining insights into the different procedural design approaches taken by the authors. You should, therefore, be able to reproduce the images presented in this book and extend the procedures presented to produce a myriad of effects.

Darwyn Peachey describes how to build procedural textures in Chapter 2. This discussion includes two-dimensional texturing and solid texturing. Two important procedures that are used throughout the book are described in this chapter: *noise* and *turbulence*. Aliasing problems of procedural techniques are described, and several antialiasing techniques suitable for procedural approaches are presented and compared.

Real-time issues for procedural texturing and shading are described by William R. Mark in Chapter 3. This chapter also provides an overview of the Stanford real-time shading language, a high-level interface for writing real-time shaders.

The design of procedural textures is also the subject of Chapters 4, 5, and 6 by Steve Worley. Chapter 4 contains useful guidelines and tricks to produce interesting textures efficiently. Chapter 5 describes some additional approaches to antialiasing, and Chapter 6 describes cellular texturing for generating interesting procedural textures.

The next four chapters by David Ebert describe how turbulence and solid texturing can be extended to produce images and animations of three-dimensional

Team LRN

gases (particulate volumes). Chapter 7 concentrates on techniques to produce still images of volumetric "gases" (steam, fog, smoke). Chapter 8 discusses how to animate these three-dimensional volumes, as well as solid textures and hypertextures. Chapter 9 describes how to extend these volumetric procedural models to model and animate realistic clouds, and Chapter 10 discusses more real-time issues for hardware-accelerated implementation of procedural textures and models.

Chapters 11 and 13 by John Hart discuss alternative approaches for procedural models and textures. Chapter 11 concentrates on procedural models of geometry, including procedural geometric instancing and grammar-based modeling. Chapter 13 concentrates on procedural textures using the texture atlas approach.

Chapter 12 by Ken Perlin discusses other types of volumetric procedural objects, such as hypertextures and surflets. Ken also gives a detailed explanation of his famous *noise* function and its implementation, as well as the rendering of procedural objects and antialiasing. Ken also describes the use of high-level, nondeterministic "texture" to create gestural motions and facial animation for synthetic actors. By blending textural controls, the apparent emotional state of a synthetic actor can be created.

Chapters 14 through 20 by Ken Musgrave (Chapter 18 is also co-authored by Larry Gritz and Steve Worley) describe fractals and their use in creating realistic landscapes, planets, and atmospherics. Ken begins by giving a brief introduction to fractals and then discusses fractal textures, landscapes, and planets. The discussion proceeds to procedural rendering techniques, genetic procedural textures, and atmospheric models.

Finally, Chapter 21 by Ken Musgrave discusses the role of procedural techniques in the human-computer interface.

Team LRN

2

Team LRN

BUILDING PROCEDURAL TEXTURES

DARWYN PEACHEY

This chapter describes how to construct procedural texture functions in a variety of ways, starting from very simple textures and eventually moving on to quite elaborate ones. The discussion is intended to give you a thorough understanding of the major building blocks of procedural textures and the ways in which they can be combined.

Geometric calculations are fundamental to procedural texturing, as they are to most of 3D computer graphics. You should be familiar with 2D and 3D points, vectors, Cartesian coordinate systems, dot products, cross products, and homogeneous transformation matrices. You should also be familiar with the RGB (red, green, blue) representation of colors and with simple diffuse and specular shading models. Consult a graphics textbook (e.g., Foley et al. 1990; Hearn and Baker 1986) if any of this sounds new or unfamiliar.

INTRODUCTION

Throughout the short history of computer graphics, researchers have sought to improve the realism of their synthetic images by finding better ways to render the appearance of surfaces. This work can be divided into *shading* and *texturing*. Shading is the process of calculating the color of a pixel or shading sample from user-specified surface properties and the shading model. Texturing is a method of varying the surface properties from point to point in order to give the appearance of surface detail that is not actually present in the geometry of the surface.

Shading models (sometimes called *illumination models, lighting models,* or *reflection models*) simulate the interaction of light with surface materials. Shading models are usually based on physics, but they always make a great number of simplifying assumptions. Fully detailed physical models would be overkill for most computer graphics purposes and would involve intractable calculations.

The simplest realistic shading model, and the one that was used first in computer graphics, is the diffuse model, sometimes called the Lambertian model. A diffuse

Team LRN

surface has a dull or matte appearance. The diffuse model was followed by a variety of more realistic shading models that simulate specular (mirrorlike) reflection (Phong 1975; Blinn 1977; Cook and Torrance 1981; He et al. 1991). Kajiya (1985) introduced anisotropic shading models in which specular reflection properties are different in different directions. Cabral, Max, and Springmeyer (1987), Miller (1988a), and Poulin and Fournier (1990) have done further research on anisotropic shading models.

All of the shading models described above are so-called local models, which deal only with light arriving at the surface directly from light sources. In the early 1980s, most research on shading models turned to the problem of simulating *global* illumination effects, which result from indirect lighting due to reflection, refraction, and scattering of light from other surfaces or participating media in the scene. Ray-tracing and radiosity techniques typically are used to simulate global illumination effects.

Texture

At about the same time as the early specular reflection models were being formulated, Catmull (1974) generated the first textured computer graphics images. Catmull's surfaces were represented as parametric patches. Each point on the 3D surface of a parametric patch corresponds to a particular 2D point (u, v) in parameter space. This 2D-to-3D correspondence implies that a two-dimensional texture image can easily be mapped onto the 3D surface. The (u, v) parameters of any point on the patch can be used to compute a corresponding pixel location in the texture image.

For example, let's assume that you have a JPEG image with a resolution of 1024 × 512 pixels, and a patch (u, v) space that extends from 0 to 1 in each dimension. For any (u, v) point on the patch surface, you can compute the corresponding pixel location in the JPEG image by simply multiplying u times 1024 and v times 512. You can use the color of the image pixel to determine the shading of the patch surface at (u, v). In this way, the image can be *texture mapped* onto the patch. (Unfortunately, this simple point-sampling approach to texture mapping usually will result in so-called aliasing artifacts. To avoid such problems, more elaborate sampling and filtering methods are needed to accurately compute the contribution of each texture image pixel to each pixel of the final rendered graphics image.)

These first efforts proved that texture provided an interesting and detailed surface appearance, instead of a simple and boring surface of a single uniform color. It was clear that texture gave a quantum leap in realism at a very low cost in human

Team LRN

effort and computer time. Variations and improvements on the notion of texture mapping quickly followed.

Blinn and Newell (1976) introduced *reflection mapping* (also called *environment mapping*) to simulate reflections from mirrorlike surfaces. Reflection mapping is a simplified form of ray tracing. The reflection mapping procedure calculates the reflection direction R of a ray from the camera or viewer to the point being shaded:

$$R = 2(N \cdot V)N - V$$

where N is the surface normal and V points toward the viewer (both must be normalized). The texture image can be accessed using the "latitude" and "longitude" angles of the normalized vector $R = (x, y, z)$:

$$\theta = \tan^{-1}(y/x)$$
$$\varphi = \sin^{-1} z$$

suitably scaled and translated to fit the range of texture image pixel coordinates. If the reflection texture is chosen carefully, the texture-mapped surface appears to be reflecting an image of its surroundings. The illusion is enhanced if both the position of the shaded point and the reflection direction are used to calculate a ray intersection with an imaginary environment sphere surrounding the scene. The latitude and longitude of the intersection can be used to access the reflection texture.

Blinn (1978) introduced *bump mapping,* which made it possible to simulate the appearance of surface bumps without actually modifying the geometry. Bump mapping uses a texture pattern to modify the direction of surface normal vectors. When the resulting normals are used in the shading calculation, the rendered surface appears to have bumps and indentations. Such bumps aren't visible at the silhouette edges of the surface, since they consist only of shading variations, not geometry.

Cook (1984) described an extension of bump mapping, called *displacement mapping,* in which textures are used actually to move the surface, not just to change the normals. Moving the surface does change the normals as well, so displacement mapping often looks very much like bump mapping except that the bumps created by displacement are even visible on the silhouettes of objects.

Reeves, Salesin, and Cook (1987) presented an algorithm for producing antialiased shadows using an image texture based on a depth image of a scene rendered from the position of the light source. A stochastic sampling technique called "percentage closer filtering" was applied to reduce the effects of aliasing in the depth image.

Team LRN

Peachey (1985) and Perlin (1985) described space-filling textures called *solid textures* as an alternative to the 2D texture images that had traditionally been used. Gardner (1984, 1985) used solid textures generated by sums of sinusoidal functions to add texture to models of trees, terrains, and clouds. Solid textures are evaluated based on the 3D coordinates of the point being textured, rather than the 2D surface parameters of the point. Consequently, solid textures are unaffected by distortions of the surface parameter space, such as you might see near the poles of a sphere. Continuity between the surface parameterization of adjacent patches isn't a concern either. The solid texture will remain consistent and have features of constant size regardless of distortions in the surface coordinate systems.

For example, objects machined from solid wood exhibit different grain textures depending on the orientation of the surface with respect to the longitudinal growth axis of the original tree. Ordinary 2D techniques of applying wood-grain textures typically result in a veneer or "plastic wood" effect. Although each surface of a block may resemble solid wood, the unrealistic relationship between the textures on the adjacent surfaces destroys the illusion. A solid texture that simulates wood gives consistent textures on all surfaces of an object (Figure 2.1).

Recent work has focused on the problem of generating a smooth texture coordinate mapping over a complex surface, such that the mapping introduces a minimal amount of distortion in the size and shape of 2D textures applied with it. Ideally, a mapping preserves distances, areas, and angles, so that the texture is undistorted. However, in practice none of these properties is easily preserved when the textured surface is complex in shape and topology. Piponi and Borshukov (2000) describe a method called "pelting" that generates a seamless texture mapping of an arbitrary

FIGURE 2.1 A wood-grain solid texture.

Team LRN

subdivision surface model. Pedersen (1995) lets the user interactively position texture patches on a surface, while helping the user minimize the distortion of the mapping. Praun, Finkelstein, and Hoppe (2000) introduce "lapped textures," in which many small swatches of texture are positioned on a surface in an overlapping fashion, according to a user-defined vector field that guides the orientation and scale of the swatches. Both Turk (2001) and Wei and Levoy (2001) generate a synthetic texture from samples and "grow" the synthetic texture right on the target surface, taking into account its shape and parameterization.

Procedural Texture

From the earliest days of texture mapping, a variety of researchers used synthetic texture models to generate texture images instead of scanning or painting them. Blinn and Newell (1976) used Fourier synthesis. Fu and Lu (1978) proposed a syntactic grammar-based texture generation technique. Schacter and Ahuja (1979) and Schacter (1980) used Fourier synthesis and stochastic models of various kinds to generate texture imagery for flight simulators. Fournier, Fussell, and Carpenter (1982) and Haruyama and Barsky (1984) proposed using stochastic subdivision ("fractal") methods to generate textures. Other researchers developed statistical texture models that analyzed the properties of natural textures and then reproduced the textures from the statistical data (Gagalowicz and Ma 1985; Garber 1981).

Cook (1984) described the "shade trees" system, which was one of the first systems in which it was convenient to generate procedural textures during rendering. Shade trees enable the use of a different shading model for each surface as well as for light sources and for attenuation through the atmosphere. Because the inputs to the shading model can be manipulated procedurally, shade trees make it possible to use texture to control any part of the shading calculation. Color and transparency textures, reflection mapping, bump mapping, displacement mapping, and solid texturing can all be implemented using shade trees.

Perlin (1985) described a complete procedural texture generation language and laid the foundation for the most popular class of procedural textures in use today, namely, those based on *noise,* a stochastic texture generation primitive.

Turk (1991) and Witkin and Kass (1991) described synthetic texture models inspired by the biochemical processes that produce (among other effects) pigmentation patterns in the skins of animals.

Sims (1991a) described a very novel texture synthesis system in which procedural textures represented as LISP expressions are automatically modified and combined by a genetic programming system. By interactively selecting among the

Team LRN

resulting textures, the user of the system can direct the simulated evolution of a texture in some desired direction.

All of the texture synthesis methods mentioned in this section might be called "procedural." But just what *is* a procedural texture?

Procedural versus Nonprocedural

The definition of procedural texture is surprisingly slippery. The adjective *procedural* is used in computer science to distinguish entities that are described by program code rather than by data structures. For instance, in artificial intelligence there is a distinction between procedural representations of knowledge and declarative ones (see, for example, section 7.3 in Rich 1983). But anything we do with computers has a procedural aspect at some level, and almost every procedure takes some parameters or inputs that can be viewed as the declarative part of the description. In the mapping of a texture image onto a surface, the procedural component is the renderer's texture mapping module, and the declarative component is the texture image.

It is tempting to define a procedural texture as one that is changed primarily by modifying the algorithm rather than by changing its parameters or inputs. However, a procedural texture in a black box is still a procedural texture, even though you might be prevented from changing its code. This is true of procedural textures that are provided to you in a non-source-code form as part of a proprietary commercial renderer or texture package. Some rendering systems allow the user to create new procedural textures and modify existing procedural textures, but many others do not.

One major defining characteristic of a procedural texture is that it is *synthetic*—generated from a program or model rather than just a digitized or painted image. But image textures can be included among procedural textures in a procedural texture language that incorporates image-based texture mapping as one of its primitive operations. Some very nice procedural textures can be based on the procedural combination, modification, or distortion of image textures. The question "How procedural are such textures?" is difficult to answer and hinges on the apparent amount of difference between the source images and the resulting texture.

Implicit and Explicit Procedures

We can distinguish two major types of procedural texturing or modeling methods: *explicit* and *implicit* methods. In explicit methods, the procedure directly generates the points that make up a shape. In implicit methods, the procedure answers a query

Team LRN

about a particular point. The most common form of implicit method is the *isocurve* (in 2D) or *isosurface* (in 3D) method. A texture pattern is defined as a function F of points P in the texture space, and the pattern consists of a level set of F, that is, the set of all points at which the function has a particular value C: $\{P \mid F(P) = C\}$. For example, a simple definition of a unit circle is the isocurve model $\{P \in R^2 | P_x^2 + P_y^2 = 1\}$. Note that F must be reasonably well behaved if the function is to form a sensible pattern: we want F to be continuous and perhaps differentiable depending on the application.

Implicit geometric models traditionally have been popular in ray tracers because the problem of intersecting a ray with the model can be expressed elegantly for implicit models: given a model $F(P) = 0$ and a ray $R(t) = O + t\,D$ with origin O and direction D, the intersection point is simply $R(t_{hit})$ where t_{hit} is the smallest positive root of $F(R(t)) = 0$. On the other hand, explicit models are convenient for depth buffer renderers (Z-buffers and A-buffers) because the explicit model can directly place points into the depth buffer in arbitrary order as the model is evaluated.

In the texturing domain, implicit procedural methods seem to be best for textures that are evaluated during rendering. In both ray tracers and depth buffer renderers, texture samples usually must be evaluated in an order that is determined by the renderer, not by the texture procedure. An implicit procedure fits perfectly in such an environment because it is designed to answer a query about any point in the texture at any time. An explicit procedure wants to generate its texture pattern in some fixed order, which probably doesn't match the needs of the rendering algorithm. In most renderers, using an explicit texture routine would require running the texture procedure as a prepass and having it generate the texture image into an image buffer, where it could be looked up as necessary for texture application during rendering. Many of the advantages of procedural textures are lost if the texture must be evaluated in a prepass.

In principle the explicit and implicit methods can be used to produce the same class of texture patterns or geometric models (virtually anything), but in practice each approach has its own class of models that are convenient or feasible. Explicit models are convenient for polygons and parametric curves and patches. Implicit models are convenient for quadrics and for patterns that result from potential or force fields. Since implicit models tend to be continuous throughout a region of the modeling space, they are appropriate for continuous density and flow phenomena such as natural stone textures, clouds, and fog.

The remainder of this chapter focuses on building procedural textures that are evaluated during rendering and, therefore, on implicit procedural textures. Some of the texture synthesis methods mentioned earlier, for example, reaction-diffusion

Team LRN

textures or syntactically generated textures, can be very difficult to generate implicitly during rendering. Other methods, such as Fourier spectral synthesis, fit well into an implicit procedural system.

Advantages of Procedural Texture

The advantages of a procedural texture over an image texture are as follows:

- A procedural representation is extremely compact. The size of a procedural texture is usually measured in kilobytes, while the size of a texture image is usually measured in megabytes. This is especially true for solid textures, since 3D texture "images" are extremely large. Nonetheless, some people have used tomographic X-ray scanners to obtain digitized volume images for use as solid textures.
- A procedural representation has no fixed resolution. In most cases it can provide a fully detailed texture no matter how closely you look at it (no matter how high the resolution).
- A procedural representation covers no fixed area. In other words, it is unlimited in extent and can cover an arbitrarily large area without seams and without unwanted repetition of the texture pattern.
- A procedural texture can be parameterized, so it can generate a class of related textures rather than being limited to one fixed texture image.

Many of these advantages are only *potential* advantages; procedural texturing gives you the tools to gain these advantages, but you must make an effort to use them. A badly written procedural texture could sacrifice any of these potential advantages. A procedural texture evaluated before rendering has only one of these advantages, namely, that it can be parameterized to generate a variety of related textures.

Disadvantages of Procedural Texture

The disadvantages of a procedural texture as compared to an image texure are as follows:

- A procedural texture can be difficult to build and debug. Programming is often hard, and programming an implicit pattern description is especially hard in nontrivial cases.

Team LRN

- A procedural texture can be a surprise. It is often easier to predict the outcome when you scan or paint a texture image. Some people choose to like this property of procedural textures and call it "serendipity." Some people hate it and say that procedural textures are hard to control.
- Evaluating a procedural texture can be slower than accessing a stored texture image. This is the classic time versus space trade-off.
- Aliasing can be a problem in procedural textures. Antialiasing can be tricky and is less likely to be taken care of automatically than it is in image-based texturing.

The RenderMan Shading Language

Some renderers provide special-purpose shading languages in which program code for shading models and procedural textures can be written. The RenderMan shading language is perhaps the best known and most widely used of such languages (Pixar 1989; Hanrahan and Lawson 1990). It is a descendant of the shade trees system described by Cook (1984). The syntax of the language is C-like, but the shading language contains built-in data types and operations that are convenient for computer graphics calculations: for example, data types for points and colors, and operators for common vector operations. The shading language lets us program any aspect of the shading calculations performed by the RenderMan renderer: surface shading, light source description, atmospheric effects, and surface displacement. Shading parameters can be made to vary across the surface to generate procedural texture effects. These could consist of variations in color, transparency, surface position, surface normal, shininess, shading model, or just about anything else you can think of.

The shading language is based on an implicit programming model, in which shading procedures called "shaders" are asked to supply the color, opacity, and other properties of specified points on a surface. As discussed in the previous section, this shading paradigm is used in most renderers, both depth buffers and ray tracers. We refer to the surface point being shaded as the "shading sample point" or simply as the "sample point." The color and opacity information that is computed by the shader for the sample point sometimes is called the "shading sample."

In the remainder of this chapter, the RenderMan shading language is used for the procedural texturing examples. You should be able to understand the examples without RenderMan experience, but if you plan to write RenderMan shaders yourself, you will benefit from reading Upstill (1990).

Team LRN

Figures 2.2–2.7 are examples of images produced at Pixar for television commercials and films. All of the textures and lighting effects in these images were generated in the RenderMan shading language. Scanned texture images were used occasionally for product packaging materials and other graphic designs that contain text, since the shapes of letters are rather tedious to produce in an implicit procedural texture. Most of the textures are purely procedural. These examples demonstrate that the class of textures that can be generated procedurally is very large indeed.

What If You Don't Use RenderMan?

We hope that this chapter will be useful to a wide audience of people interested in procedural textures, not only to those who write RenderMan shaders. The

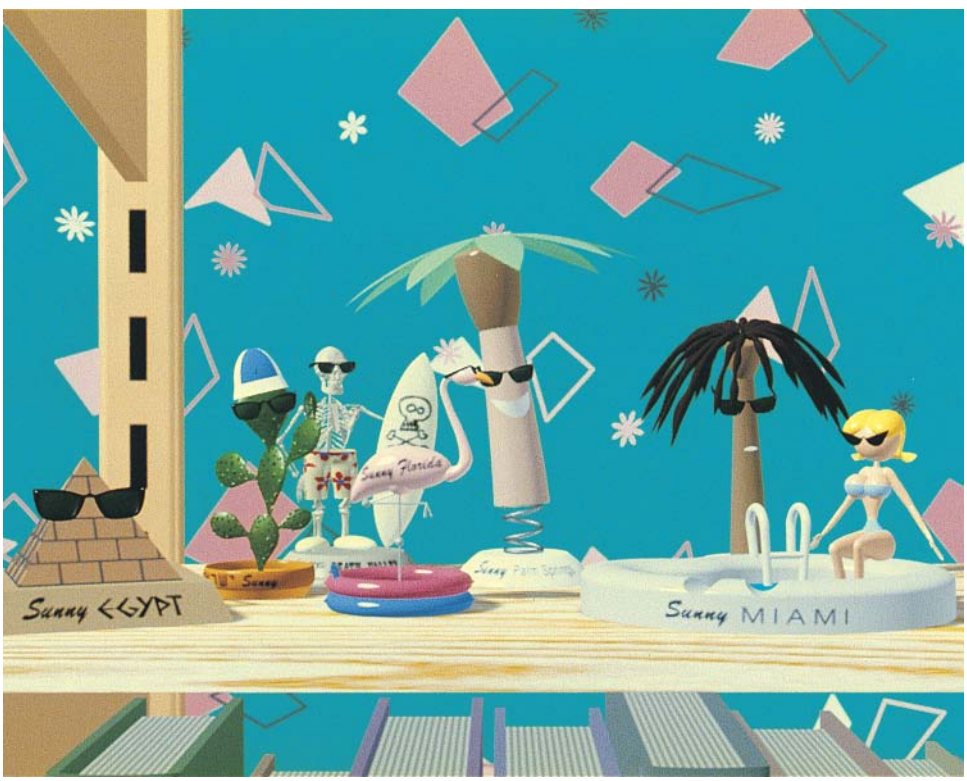


FIGURE 2.2 *Knickknack*. Copyright © 1989 Pixar.

Team LRN

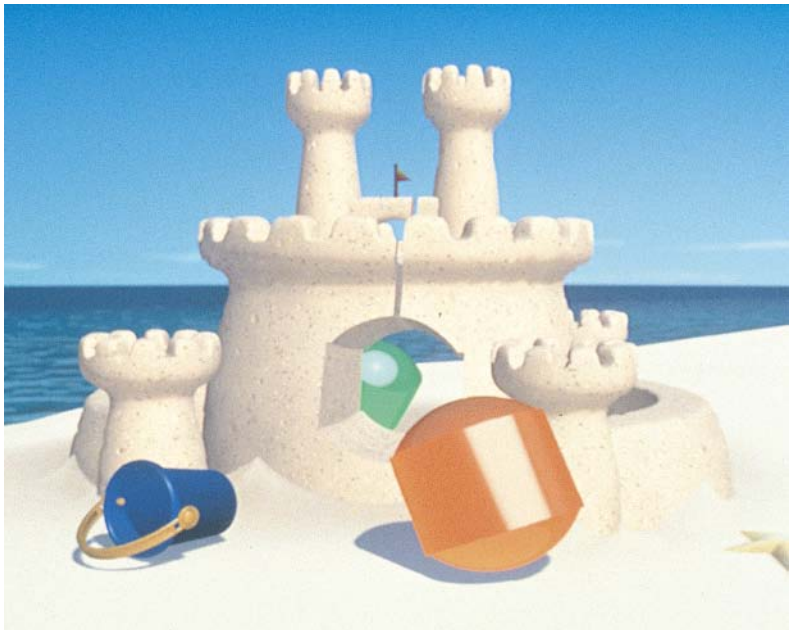


FIGURE 2.3 Lifesavers, "At the Beach." Image by Pixar. Copyright © 1991 FCB/Leber Katz Partners.

FIGURE 2.4 Carefree Gum, "Bursting." Image by Pixar. Copyright © 1993 FCB/Leber Katz Partners.

Team LRN

FIGURE 2.5 Listerine, "Knight." Image by Pixar. Copyright © 1991 J. Walter Thompson.

FIGURE 2.6 Listerine, "Arrows." Image by Pixar. Copyright © 1994 J. Walter Thompson.

FIGURE 2.7 Listerine, "Arrows." Image by Pixar. Copyright © 1994 J. Walter Thompson.

RenderMan shading language is a convenient way to express procedural shaders, and it allows us to easily render high-quality images of the examples. However, this chapter also describes the C language implementation of many of the procedural texture building-block functions that are built into the RenderMan shading language. This should allow you to translate the shading language examples into C code that you can use in your own rendering program. The RenderMan shading language is superficially much like the C language, so you might have to look closely to see whether a given fragment of code is in C or in the shading language.

The RenderMan shading language provides functions that access image textures. It is not practical to include an implementation of those functions in this chapter. Efficient image texture mapping with proper filtering is a complex task, and the program code consists of several thousand lines of C. Most other renderers have their own texture functions, and you should be able to translate the RenderMan texture functions into the appropriate code to access the texture functions of your renderer.

Team LRN

PROCEDURAL PATTERN GENERATION

Sitting down to build a procedural texture can be a daunting experience, at least the first few times. This section gives some hints, guidelines, and examples that will help you stave off the anxiety of facing an empty text editor screen with no idea of how to begin. It is usually much easier to start by copying an example and modifying it to do what you want than it is to start from scratch. Most of the time, it is better to borrow code than to invent it.

Most surface shaders can be split into two components called *pattern generation* and the *shading model*. Pattern generation defines the texture pattern and sets the values of surface properties that are used by the shading model. The shading model simulates the behavior of the surface material with respect to diffuse and specular reflection.

Shading Models

Most surface shaders use one of a small number of shading models. The most common model includes diffuse and specular reflection and is called the "plastic" shading model. It is expressed in the RenderMan shading language as follows:

```
surface
plastic(float Ka = 1, Kd = 0.5, Ks = 0.5;
      float roughness = 0.1;
      color specularcolor = color (1,1,1))
{
      point Nf = faceforward(normalize(N), I);
      point V = normalize(-I);
      Oi = Os;
      Ci = Os * (Cs * (Ka * ambient()
            + Kd * diffuse(Nf))
            + specularcolor * Ks
                  * specular(Nf, V, roughness));
}
```

The following paragraphs explain the workings of the `plastic` shader in detail, as a way of introducing the RenderMan shading language to readers who are not familiar with it.

The parameters of the `plastic` shader are the coefficients of ambient, diffuse, and specular reflectance; the roughness, which controls the size of specular highlights; and the color of the specular highlights. Colors are represented by RGB

triples, specifying the intensities of the red, green, and blue primary colors as numbers between 0 and 1. For example, in this notation, `color (1,1,1)` is white.

Any RenderMan surface shader can reference a large collection of built-in quantities such as `P`, the 3D coordinates of the point on the surface being shaded, and `N`, the surface normal at `P`. The normal vector is perpendicular to the tangent plane of the surface at `P` and points toward the outside of the surface. Because surfaces can be two-sided, it is possible to see the inside of a surface; in that case we want the normal vector to point toward the camera, not away from it. The built-in function `faceforward` simply compares the direction of the incident ray vector `I` with the direction of the normal vector `N`. `I` is the vector from the camera position to the point `P`. If the two vectors `I` and `N` point in the same direction (i.e., if their dot product is positive), `faceforward` returns `-N` instead of `N`.

The first statement in the body of the shader declares and initializes a surface normal vector `Nf`, which is normalized and faces toward the camera. The second statement declares and initializes a "viewer" vector `V` that is normalized and gives the direction to the camera. The third statement sets the output opacity `Oi` to be equal to the input surface opacity `Os`. If the surface is somewhat transparent, the opacity is less than one. Actually, `Os` is a color, an RGB triple that gives the opacity of the surface for each of the three primary colors. For an opaque surface, `Os` is `color(1,1,1)`.

The final statement in the shader does the interesting work. The output color `Ci` is set to the product of the opacity and a color.[1] The color is the sum of an ambient term and a diffuse term multiplied by the input surface color `Cs`, added to a specular term whose color is determined by the parameter `specularcolor`. The built-in functions `ambient`, `diffuse`, and `specular` gather up all of the light from multiple light sources according to a particular reflection model. For instance, `diffuse` computes the sum of the intensity of each light source multiplied by the dot product of the direction to the light source and the surface normal `Nf` (which is passed as a parameter to `diffuse`).

The plastic shading model is flexible enough to include the other two most common RenderMan shading models, the "matte" model and the "metal" model, as special cases. The matte model is a perfectly diffuse reflector, which is equivalent to plastic with a `Kd` of 1 and a `Ks` of 0. The metal model is a perfectly specular reflector,

1. The color is multiplied by the opacity because RenderMan uses an "alpha blending" technique to combine the colors of partially transparent surfaces, similar to the method described by Porter and Duff (1984).

Team LRN

which is equivalent to plastic with a `Kd` of 0, a `Ks` of 1, and a `specularcolor` the same as `Cs`. The `specularcolor` parameter is important because it has been observed that when illuminated by a white light source, plastics and other dielectric (insulating) materials have white highlights while metals and other conductive materials have colored highlights (Cook and Torrance 1981). For example, gold has a gold-colored highlight.

The `plastic` shader is a good starting point for many procedural texture shaders. We will simply replace the `Cs` in the last statement of the shader with a new color variable `Ct`, the texture color that is computed by the pattern generation part of the shader.

Pattern Generation

Pattern generation is usually the hard part of writing a RenderMan shader because it involves figuring out how to generate a particular texture pattern procedurally.

If the texture pattern is simply an image texture, the shader can call the built-in shading language function `texture`:

```
Ct = texture("name.tx",s,t);
```

The shading language `texture` function looks up pixel values from the specified image texture "`name.tx`" and performs filtering calculations as needed to prevent aliasing artifacts. The `texture` function has the usual 2D texture space with the texture image in the unit square. The built-in variables `s` and `t` are the standard RenderMan texture coordinates, which by default range over the unit interval [0, 1] for any type of surface. The shading language also provides an `environment` function whose 2D texture space is accessed using a 3D direction vector that is converted internally into 2D form to access a latitude-longitude or cube-face environment map (Greene 1986).

When the texture image is suitable, there is no easier or more realistic way to generate texture patterns. Unfortunately, it is difficult to get a texture image that is suitable for many texture applications. It would be nice if all desired textures could be implemented by finding a sample of the actual material, photographing it, and scanning the photograph to produce a beautiful texture image. But this approach is rarely adequate by itself. If a material is not completely flat and smooth, the photograph will capture information about the lighting direction and the light source. Each bump in the material will be shaded based on its slope, and in the worst case, the bumps will cast shadows. Even if the material is flat and smooth, the photograph often will record uneven lighting conditions, reflections of the environment

Team LRN

surrounding the material, highlights from the light sources, and so on. This information generates incorrect visual cues when the photograph is texture mapped onto a surface in a scene with simulated lighting and environmental characteristics that differ from those in the photograph. A beautiful photograph often looks out of place when texture mapped onto a computer graphics model.

Another problem with photographic texture images is that they aren't infinitely large. When a large area must be covered, copies of the texture are placed side by side. A seam is usually visible because the texture pixels don't match from the top to the bottom or the left to the right. Sometimes retouching can be used to make the edges match up seamlessly. Even when this has been done, a large area textured with many copies of the image can look bad because it is obvious that a small amount of texture data has been used to texture a large area. Prominent features in the texture map are replicated over and over in an obviously repetitive way. Such problems can be avoided by making sure that the texture photograph covers a large area of the texture, but this will result in visible pixel artifacts in a rendered image that magnifies a tiny part of the texture. (As an alternative approach, you might consider the remarkably successful algorithm described by Wei and Levoy (2000), which analyzes a small texture image and then synthesizes a larger texture image that resembles the smaller image, but does not duplicate it.)

The right way to use image textures is to design the shader first and then go out and get a suitable texture photograph to scan. The lighting, the size of the area photographed, and the resolution of the scanning should all be selected based on the application of the shader. In many cases the required texture images will be bump altitude maps, monochrome images of prominent textural features, and so on. The texture images may be painted by hand, generated procedurally, or produced from scanned material after substantial image processing and retouching.

Procedural pattern generators are more difficult to write. In addition to the problems of creating a small piece of program code that produces a convincing simulation of some material, the procedural pattern generator must be antialiased to prevent fine details in the pattern from aliasing when seen from far away.

Procedural pattern generators are harder to write than texture image shaders, but they have several nice properties. It is usually easy to make the texture cover an arbitrarily large area without seams or objectionable repetition. It is easy to separate small-scale shape effects such as bumps and dimples from color variations; each can be generated separately in the procedural shader.

Writing procedural pattern generators is still an art form; there is no recipe that will work every time. This is a programming task in which the problem is to generate the appearance of some real-world material. The first step is to go out and examine

Team LRN

the real material: its color and color variations, its reflection properties, and its surface characteristics (smooth or rough, bumpy or pitted). Photographs that you can take back to your desk are very valuable. Architectural and design magazines and books are a good source of pictures of materials.

Texture Spaces

The RenderMan shading language provides many different built-in coordinate systems (also called *spaces)*. A coordinate system is defined by the concatenated stack of transformation matrices that is in effect at a given point in the hierarchical structure of the RenderMan geometric model.

- The `current` space is the one in which shading calculations are normally done. In most renderers, `current` space will turn out to be either `camera` space or `world` space, but you shouldn't depend on this.
- The `world` space is the coordinate system in which the overall layout of your scene is defined. It is the starting point for all other spaces.
- The `object` space is the one in which the surface being shaded was defined. For instance, if the shader is shading a sphere, the `object` space of the sphere is the coordinate system that was in effect when the `RiSphere` call was made to create the sphere. Note that an object made up of several surfaces all using the same shader might have different object spaces for each of the surfaces if there are geometric transformations between the surfaces.
- The `shader` space is the coordinate system that existed when the shader was invoked (e.g., by an `RiSurface` call). This is a very useful space because it can be attached to a user-defined collection of surfaces at an appropriate point in the hierarchy of the geometric model so that all of the related surfaces share the same `shader` space.

In addition, user-defined coordinate systems can be created and given names using the `RiCoordinateSystem` call. These coordinate systems can be referenced by name in the shading language.

It is very important to choose the right texture space when defining your texture. Using the 2D surface texture coordinates (s, t) or the surface parameters (u, v) is fairly safe, but might cause problems due to nonuniformities in the scale of the parameter space (e.g., compression of the parameter space at the poles of a sphere). Solid textures avoid that problem because they are defined in terms of the 3D coordinates of the sample point. If a solid texture is based on the `camera` space coordinates

Team LRN

of the point, the texture on a surface will change whenever either the camera or the object is moved. If the texture is based on `world` space coordinates, it will change whenever the object is moved. In most cases, solid textures should be based on the `shader` space coordinates of the shading samples, so that the texture will move properly with the object. The `shader` space is defined when the shader is invoked, and that can be done at a suitable place in the transformation hierarchy of the model so that everything works out.

It is a simplification to say that a texture is defined in terms of a single texture space. In general a texture is a combination of a number of separate "features," each of which might be defined in terms of its own *feature space*. If the various feature spaces that are used in creating the texture are not based on one underlying texture space, great care must be exercised to be sure that texture features don't shift with respect to one another. The feature spaces should have a fixed relationship that doesn't change when the camera or the object moves.

Layering and Composition

The best approach to writing a complex texture pattern generator is to build it up from simple parts. There are a number of ways to combine simple patterns to make complex patterns.

One technique is *layering,* in which simple patterns are placed on top of one another. For example, the colors of two texture layers could be added together. Usually, it is better to have some texture function control how the layers are combined. The shading language `mix` function is a convenient way of doing this.

```
C = mix(C0, C1, f);
```

The number `f`, between 0 and 1, is used to select one of the colors `C0` and `C1`. If `f` is 0, the result of the `mix` is `C0`. If `f` is 1, the result is `C1`. If `f` is between 0 and 1, the result is a linearly interpolated mixture of `C0` and `C1`. The `mix` function is defined as

```
color
mix(color C0, color C1, float f)
{
      return (1-f)*C0 + f*C1;
}
```

`C0` and `C1` can be fixed colors, or they can be two subtextures. In either case, they are combined under the control of the number `f`, which is itself the result of some procedural texture function.

Team LRN

When two colors are multiplied together in the shading language, the result is a color whose RGB components are the product of the corresponding components from the input colors. That is, the red result is the product of the red components of the two inputs. Color multiplication can simulate the filtering of one color by the other. If color `C0` represents the transparency of a filter to red, green, and blue light, then `C0*C1` represents the color `C1` as viewed through the filter.

Be careful when using a four-channel image texture that was created from an RGBA image (an image with an opacity or "alpha" channel) because the colors in such an image are normally premultiplied by the value of the alpha channel. In this case, it is not correct simply to combine the RGB channels with another color under control of the alpha channel. The correct way to merge an RGBA texture over another texture color `Ct` is

```
color C;
float A;

C = color texture("mytexture",s,t);
A = texture("mytexture"[3],s,t);
result = C + (1-A) * Ct;
```

`C` is the image texture color, and `A` is the alpha channel of the image texture (channel number 3). Since `C` has already been multiplied by `A`, the expression `C + (1–A*Ct)` is the right way to *lerp*[2] between `C` and `Ct`.

Another way to combine simple functions to make complex functions is *functional composition,* using the outputs of one or more simple functions as the inputs of another function. For example, one function generates a number that varies between 0 and 1 in a particular way, and this number is used as the input to another function that generates different colors for different values of its numerical input. One function might take inputs that are points in one texture space and produce output points in another space that are the input to a second function; in this case, the first function transforms points into the feature space needed by the second function. Composition is very powerful and is so fundamental to programming that you really can't avoid using it.

The computer science literature concerning *functional programming* is a good source of techniques for combining functions (Ghezzi and Jazayeri 1982). Functional languages such as LISP (Winston and Horn 1984) rely heavily on composition and related techniques.

2. In computer graphics, linear interpolation is colloquially called *lerping*.

Team LRN

The remainder of this section presents a series of primitive functions that are used as building blocks for procedural textures. The presentation includes several examples of the use of these primitives in procedural texture shaders.

Steps, Clamps, and Conditionals

From the earlier discussion on methods of combining primitive operations to make procedural patterns, it should be clear that functions taking parameters and returning values are the most convenient kind of primitive building blocks. Steps and clamps are conditional functions that give us much the same capabilities that `if` statements give us. But steps and clamps are often more convenient, simply because they are functions.

The RenderMan shading language function `step(a,x)` returns the value 0 when `x` is less than `a` and returns 1 otherwise. The `step` function can be written in C as follows:

```
float
step(float a, float x)
{
      return (float) (x >= a);
}
```

A graph of the `step` function is shown in Figure 2.8.

The main use of the `step` function is to replace an `if` statement or to produce a sharp transition between one type of texture and another type of texture. For example, an `if` statement such as

```
if (u < 0.5)
      Ci = color (1,1,.5);
else
      Ci = color ( .5,.3,1);
```

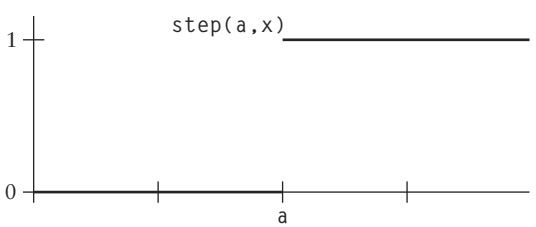


FIGURE 2.8 The `step` function.

can be rewritten to use the `step` function as follows:

```
Ci = mix(color (1,1,.5), color (.5,.3,1), step(0.5, u));
```

Later in this chapter when we examine antialiasing, you'll learn how to create an antialiased version of the `step` function. Writing a procedural texture with a lot of `if` statements instead of step functions can make antialiasing much harder.[3]

Two `step` functions can be used to make a rectangular pulse as follows:

```
#define PULSE(a,b,x) (step((a),(x)) - step((b),(x)))
```

This preprocessor macro generates a pulse that begins at `x = a` and ends at `x = b`. A graph of the pulse is shown in Figure 2.9.

The RenderMan shading language function `clamp(x,a,b)` returns the value `a` when `x` is less than `a`, the value of `x` when `x` is between `a` and `b`, and the value `b` when `x` is greater than `b`. The `clamp` function can be written in C as follows:

```
float
clamp(float x, float a, float b)
{
      return (x < a ? a: (x > b ? b : x));
}
```

A graph of the `clamp` function is shown in Figure 2.10.

The well-known `min` and `max` functions are closely related to `clamp`. In fact, `min` and `max` can be written as `clamp` calls, as follows:

```
min(x, b) ≡ clamp(x, x, b)
```

and

```
max(x, a) ≡ clamp(x, a, x)
```

Alternatively, `clamp` can be expressed in terms of `min` and `max`:

```
clamp(x, a, b) ≡ min(max(x, a), b)
```

3. Another reason for using `step` instead of `if` in RenderMan shaders is that it encourages you to compute the inputs of a conditional everywhere, not just in the fork of the conditional where they are used. This can avoid problems in applying image textures and other area operations.

Team LRN

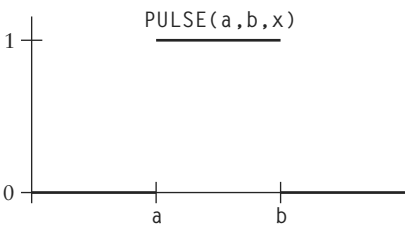


FIGURE 2.9 Two steps used to make a pulse.

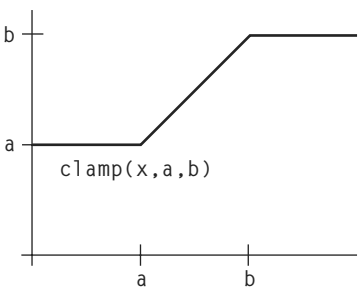


FIGURE 2.10 The `clamp` function.

For completeness, here are the C implementations of `min` and `max`:

```
float
min(float a, float b)
{
      return (a < b ? a : b);
}
float
max(float a, float b)
{
      return (a < b ? b : a);
}
```

Another special conditional function is the `abs` function, expressed in C as follows:

```
float
abs(float x)
{
      return (x < 0 ? -x : x);
}
```

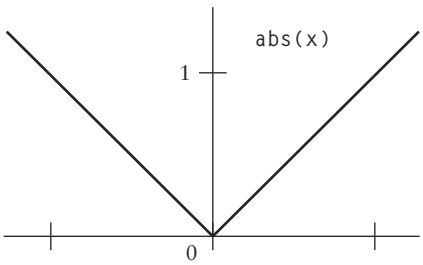


FIGURE 2.11 The `abs` function.

A graph of the `abs` function is shown in Figure 2.11. The `abs` function can be viewed as a rectifier; for example, it will convert a sine wave that oscillates between −1 and 1 into a sequence of positive sinusoidal pulses that range from 0 to 1.

In addition to the "pure" or "sharp" conditionals `step`, `clamp`, `min`, `max`, and `abs`, the RenderMan shading language provides a "smooth" conditional function called `smoothstep`. This function is similar to `step`, but instead of a sharp transition from 0 to 1 at a specified threshold, `smoothstep(a,b,x)` makes a gradual transition from 0 to 1 beginning at threshold `a` and ending at threshold `b`. In order to do this, `smoothstep` contains a cubic function whose slope is 0 at `a` and `b` and whose value is 0 at `a` and 1 at `b`. There is only one cubic function that has these properties for `a = 0` and `b = 1`, namely, the function $3x^2 - 2x^3$.

Here is a C implementation of `smoothstep`, with the cubic function expressed according to Horner's rule:[4]

```
float
smoothstep(float a, float b, float x)
{
      if (x < a)
            return 0;
      if (x >= b)
            return 1;
      x = (x - a)/(b - a);
      return (x*x * (3 - 2*x));
}
```

A graph of the `smoothstep` function is shown in Figure 2.12.

4. Horner's rule is a method of nested multiplication for efficiently evaluating polynomials.

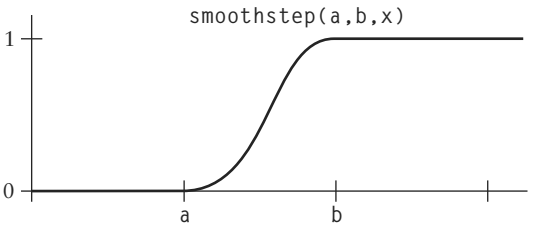


FIGURE 2.12 The `smoothstep` function.

The `smoothstep` function is used instead of `step` in many procedural textures because sharp transitions often result in unsightly artifacts. Many of the artifacts are due to aliasing, which is discussed at length later in this chapter. Sharp transitions can be very unpleasant in animated sequences because some features of the texture pattern appear suddenly as the camera or object moves (the features are said to "pop" on and off). Most of the motion in an animated sequence is carefully "eased" in and out to avoid sudden changes in speed or direction; the `smoothstep` function helps to keep the procedural textures in the scene from changing in equally unsettling ways.

Periodic Functions

The best-known periodic functions are `sin` and `cos`. They are important because of their close ties to the geometry of the circle, to angular measure, and to the representation of complex numbers. It can be shown that other functions can be built up from a sum of sinusoidal terms of different frequencies and phases (see the discussion on "Spectral Synthesis" on page 48).

`sin` and `cos` are available as built-in functions in C and in the RenderMan shading language. Some ANSI C implementations provide single-precision versions of `sin` and `cos`, called `sinf` and `cosf`, which you might prefer to use to save computation time. A graph of the `sin` and `cos` functions is shown in Figure 2.13.

Another important periodic function is the `mod` function. `mod(a,b)` gives the positive remainder obtained when dividing `a` by `b`. C users beware! Although C has a built-in integer remainder operator "%" and math library functions `fmod` and `fmodf` for `double` and `float` numbers, all of these are really remainder functions, not modulus functions, in that they will return a negative result if the first operand, `a`, is negative. Instead, you might use the following C implementation of `mod`:

Team LRN

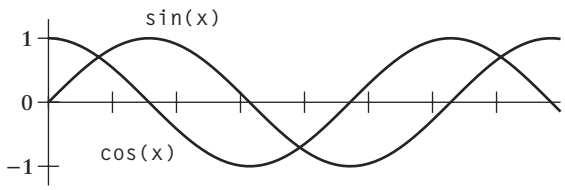


FIGURE 2.13 The `sin` and `cos` functions.

```
float
mod(float a, float b)
{
   int n = (int)(a/b);
   a -= n*b;
   if (a < 0)
     a += b;
   return a;
}
```

A graph of the periodic sawtooth function `mod(x,a)/a` is shown in Figure 2.14. This function has an amplitude of one and a period of `a`.

By applying `mod` to the inputs of some other function, we can make the other function periodic too. Take any function, say, $f(x)$, defined on the interval from 0 to 1 (technically, on the half-open interval [0, 1]). Then f (`mod(x,a)/a`) is a periodic function. To make this work out nicely, it is best if $f(0) = f(1)$ and even better if the derivatives of f are also equal at 0 and 1. For example, the pulse function `PULSE(0.4,0.6,x)` can be combined with the `mod` function to get the periodic square wave function `PULSE(0.4,0.6,mod(x,a)/a)` with its period equal to `a` (see Figure 2.15).

It's often preferable to use another `mod`-like idiom instead of `mod` in your shaders. We can think of `xf = mod(a,b)/b` as the fractional part of the ratio `a/b`. In many cases it is useful to have the integer part of the ratio, `xi`, as well.

```
float xf, xi;
xf = a/b;
xi = floor(xf);
xf -= xi;
```

The function `floor(x)` returns the largest integer that is less than or equal to `x`. Since the `floor` function is a built-in part of both the C math library and the

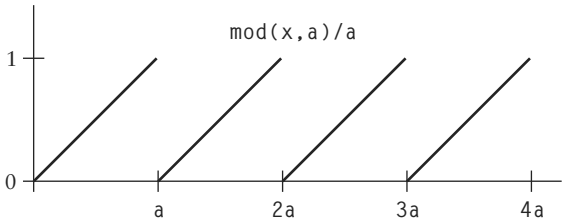


FIGURE 2.14 The periodic function `mod(x,a)/a`.

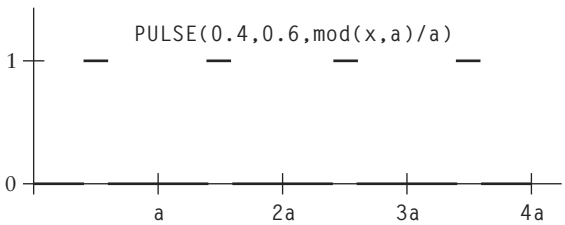


FIGURE 2.15 How to make a function periodic.

RenderMan shading language, this piece of code will work equally well in either language. Some versions of ANSI C provide a `floorf` function whose argument and result are single-precision float numbers. In C, the following macro is an alternative to the built-in `floor` function:

```
#define FLOOR(x) ((int)(x) - ((x) < 0 && (x) != (int)(x)))
```

`FLOOR` isn't precisely the same as `floor`, because `FLOOR` returns a value of `int` type rather than `double` type. Be sure that the argument passed to `FLOOR` is the name of a variable, since the macro may evaluate its argument up to four times.

A closely related function is the ceiling function `ceil(x)`, which returns the smallest integer that is greater than or equal to `x`. The function is built into the shading language and the C math library. ANSI C provides the single-precision version `ceilf`. The following macro is an alternative to the C library function:

```
#define CEIL(x) ((int)(x) + ((x) > 0 && (x) != (int)(x)))
```

Team LRN

Splines and Mappings

The RenderMan shading language has a built-in spline function, which is a one-dimensional Catmull-Rom interpolating spline through a set of so-called *knot* values. The parameter of the spline is a floating-point number.

```
result = spline(parameter,
  knot1, knot2, . . . , knotN-1, knotN);
```

In the shading language, the knots can be numbers, colors, or points (but all knots must be of the same type). The result has the same data type as the knots. If `parameter` is 0, the result is `knot2`. If `parameter` is 1, the result is `knotN-1`. For values of `parameter` between 0 and 1, the value of `result` interpolates smoothly between the values of the knots from `knot2` to `knotN-1`. The `knot1` and `knotN` values determine the derivatives of the spline at its end points. Because the spline is a cubic polynomial, there must be at least four knots.

Here is a C language implementation of `spline` in which the knots must be floating-point numbers:

```
/* Coefficients of basis matrix. */
#define CR00      -0.5
#define CR01      1.5
#define CR02      -1.5
#define CR03      0.5
#define CR10      1.0
#define CR11      -2.5
#define CR12      2.0
#define CR13      -0.5
#define CR20      -0.5
#define CR21      0.0
#define CR22      0.5
#define CR23      0.0
#define CR30      0.0
#define CR31      1.0
#define CR32      0.0
#define CR33      0.0

float
spline(float x, int nknots, float *knot)
{
      int span;
      int nspans = nknots - 3;
      float c0, c1, c2, c3; /* coefficients of the cubic.*/
      if (nspans < 1){/* illegal */
```

Team LRN

```
            fprintf(stderr, "Spline has too few knots.\n");
            return 0;
        }
        /* Find the appropriate 4-point span of the spline. */
        x = clamp(x, 0, 1) * nspans;
        span = (int) x;
        if (span >= nknots - 3)
            span = nknots - 3;
        x -= span;
        knot += span;

        /* Evaluate the span cubic at x using Horner's rule. */

        c3 = CR00*knot[0] + CR01*knot[1] + CR02*knot[2] + CR03*knot[3];
        c2 = CR10*knot[0] + CR11*knot[1] + CR12*knot[2] + CR13*knot[3];
        c1 = CR20*knot[0] + CR21*knot[1] + CR22*knot[2] + CR23*knot[3];
        c0 = CR30*knot[0] + CR31*knot[1] + CR32*knot[2] + CR33*knot[3];

        return ((c3*x + c2)*x + c1)*x + c0;

}
```

A graph of a particular example of the `spline` function is shown in Figure 2.16.

This code can easily be adapted to work with knots that are colors or points. Just do the same thing three times, once for each of the components of the knots. In other words,

```
spline(parameter, (x1,y1,z1), . . . , (xN,yN,zN))
```

is exactly equivalent to

```
(spline(parameter, x1, . . . , xN),
spline(parameter, y1, . . . , yN),
spline(parameter, z1, . . . , zN))
```

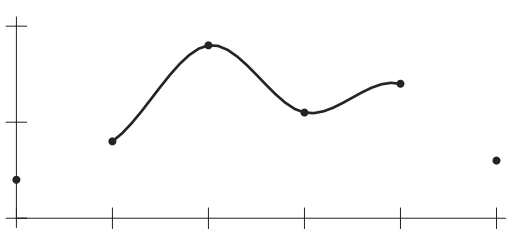


FIGURE 2.16 An example of the `spline` function.

The `spline` function is used to map a number into another number or into a color. A spline can approximate any function on the [0, 1] interval by giving values of the function at equally spaced sample points as the knots of the spline. In other words, the spline can interpolate function values from a table of known values at equally spaced values of the input parameter. A spline with colors as knots can be used as a *color map* or *color table*.

An example of this technique is a shader that simulates a shiny metallic surface by a procedural reflection map texture. The shader computes the reflection direction `R` of the viewer vector `V`. The vertical component of `R` in `world` space is used to look up a color value in a spline that goes from brown earth color below to pale bluish-white at the horizon and then to deeper shades of blue in the sky. Note that the shading language's built-in `vtransform` function properly converts a direction vector from the current rendering space to another coordinate system specified by name.

```
#define BROWN  color (0.1307,0.0609,0.0355)
#define BLUE0  color (0.4274,0.5880,0.9347)
#define BLUE1  color (0.1221,0.3794,0.9347)
#define BLUE2  color (0.1090,0.3386,0.8342)
#define BLUE3  color (0.0643,0.2571,0.6734)
#define BLUE4  color (0.0513,0.2053,0.5377)
#define BLUE5  color (0.0326,0.1591,0.4322)
#define BLACK  color (0,0,0)
surface
metallic( )
{
      point Nf = normalize(faceforward(N, I));
      point V = normalize(-I);
      point R; /* reflection direction */
      point Rworld; /* R in world space */
      color Ct;
      float altitude;

      R = 2 * Nf * (Nf . V) - V;
      Rworld = normalize(vtransform("world", R));
      altitude = 0.5 * zcomp(Rworld) + 0.5;
      Ct = spline(altitude,
            BROWN, BROWN, BROWN, BROWN, BROWN,
            BROWN, BLUE0, BLUE1, BLUE2, BLUE3,
            BLUE4, BLUE5, BLACK);
      Oi = Os;
      Ci = Os * Cs * Ct;
}
```

Figure 2.17 is an image shaded with the metallic reflection map shader.

Since `mix` functions and so many other selection functions are controlled by values that range over the [0, 1] interval, mappings from the unit interval to itself can be especially useful. Monotonically increasing functions on the unit interval can be used to change the distribution of values in the interval. The best-known example of such a function is the "gamma correction" function used to compensate for the nonlinearity of CRT display systems:

```
float
gammacorrect(float gamma, float x)
{
      return pow(x, 1/gamma);
}
```

Figure 2.18 shows the shape of the gamma correction function for `gamma` values of 0.4 and 2.3. If `x` varies over the [0, 1] interval, then the result is also in that interval. The zero and one end points of the interval are mapped to themselves. Other values are shifted upward toward one if `gamma` is greater than one, and shifted downward toward zero if `gamma` is between zero and one.

Perlin and Hoffert (1989) use a version of the gamma correction function that they call the `bias` function. The `bias` function replaces the `gamma` parameter with a parameter `b`, defined such that `bias(b,0.5) = b`.

Team LRN

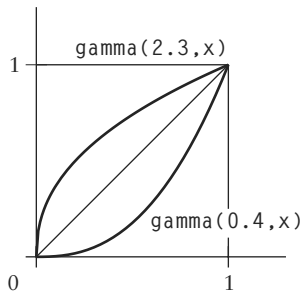


FIGURE 2.18 The gamma correction function.

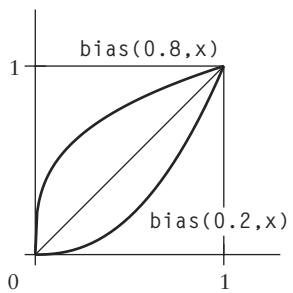


FIGURE 2.19 The `bias` function.

```
float
bias(float b, float x)
{
   return pow(x, log(b)/log(0.5));
}
```

Figure 2.19 shows the shape of the `bias` function for different choices of `b`.

Perlin and Hoffert (1989) present another function to remap the unit interval. This function is called `gain` and can be implemented as follows:

```
float
gain(float g, float x)
{
   if (x < 0.5)
     return bias(1-g, 2*x)/2;
   else
     return 1 - bias(1-g, 2 - 2*x)/2;
}
```

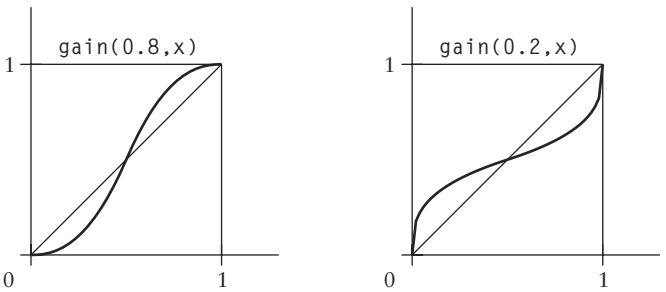


FIGURE 2.20 The `gain` function.

Regardless of the value of `g`, all `gain` functions return 0.5 when `x` is 0.5. Above and below 0.5, the `gain` function consists of two scaled-down `bias` curves forming an S-shaped curve. Figure 2.20 shows the shape of the `gain` function for different choices of `g`.

Schlick (1994) presents approximations to `bias` and `gain` that can be evaluated more quickly than the power functions given here.

Example: Brick Texture

One of the standard texture pattern clichés in computer graphics is the checkerboard pattern. This pattern was especially popular in a variety of early papers on antialiasing. Generating a checkerboard procedurally is quite easy. It is simply a matter of determining which square of the checkerboard contains the sample point and then testing the parity of the sum of the row and column to determine the color of that square.

This section presents a procedural texture generator for a simple brick pattern that is related to the checkerboard but is a bit more interesting. The pattern consists of rows of bricks in which alternate rows are offset by one-half the width of a brick. The bricks are separated by a mortar that has a different color than the bricks. Figure 2.21 is a diagram of the brick pattern.

The following is a listing of the shading language code for the brick shader, with explanatory remarks inserted here and there.

```
#define BRICKWIDTH          0.25
#define BRICKHEIGHT         0.08
#define MORTARTHICKNESS     0.01

#define BMWIDTH             (BRICKWIDTH+MORTARTHICKNESS)
#define BMHEIGHT            (BRICKHEIGHT+MORTARTHICKNESS)
```

Team LRN

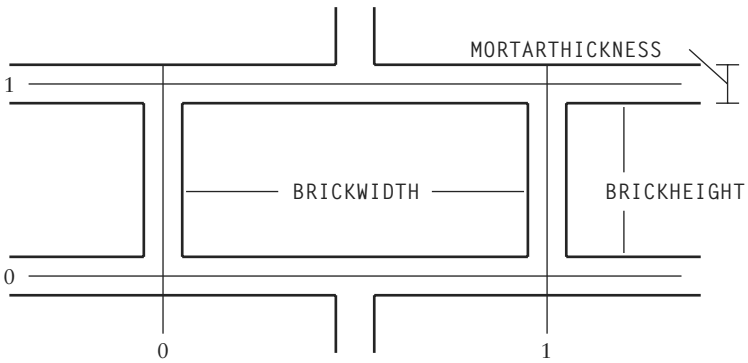


FIGURE 2.21 The geometry of a brick.

```
#define MWF                 (MORTARTHICKNESS*0.5/BMWIDTH)
#define MHF                 (MORTARTHICKNESS*0.5/BMHEIGHT)

surface brick(
    uniform float Ka = 1;
    uniform float Kd = 1;
    uniform color Cbrick = color (0.5, 0.15, 0.14);
    uniform color Cmortar = color (0.5, 0.5, 0.5);
    )
{
          color Ct;
          point Nf;
          float ss, tt, sbrick, tbrick, w, h;
          float scoord = s;
          float tcoord = t;

          Nf = normalize(faceforward(N, I));

          ss = scoord / BMWIDTH;
          tt = tcoord / BMHEIGHT;

          if (mod(tt*0.5,1) > 0.5)
          ss += 0.5; /* shift alternate rows */
```

The texture coordinates `scoord` and `tcoord` begin with the values of the standard texture coordinates `s` and `t`, and then are divided by the dimensions of a brick (including one-half of the mortar around the brick) to obtain new coordinates `ss` and `tt` that vary from 0 to 1 within a single brick. `scoord` and `tcoord` become the coordinates of the upper-left corner of the brick containing the point being shaded.

Team LRN

Alternate rows of bricks are offset by one-half brick width to simulate the usual way in which bricks are laid.

```
    sbrick = floor(ss); /* which brick? */
    tbrick = floor(tt); /* which brick? */
    ss -= sbrick;
    tt -= tbrick;
```

Having identified which brick contains the point being shaded, as well as the texture coordinates of the point within the brick, it remains to determine whether the point is in the brick proper or in the mortar between the bricks.

```
    w = step(MWF,ss) - step(1-MWF,ss);
    th = step(MHF,tt) - step(1-MHF,tt);

    Ct = mix(Cmortar, Cbrick, w*h);

    /* diffuse reflection model */
    Oi = Os;
    Ci = Os * Ct * (Ka * ambient() + Kd * diffuse(Nf));
}
```

The rectangular brick shape results from two pulses (see page 28), a horizontal pulse `w` and a vertical pulse `h`. `w` is zero when the point is horizontally within the mortar region and rises to one when the point is horizontally within the brick region. `h` does the same thing vertically. When the two values are multiplied together, the result is the logical AND of `w` and `h`. That is, `w * h` is nonzero only when the point is within the brick region both horizontally and vertically. In this case, the `mix` function switches from the mortar color `Cmortar` to the brick color `Cbrick`.

The shader ends by using the texture color `Ct` in a simple diffuse shading model to shade the surface. Figure 2.22 is an image rendered with the brick texture from this example.

Bump-Mapped Brick

Now let's try our hand at some procedural bump mapping. Recall that bump mapping involves modifying the surface normal vectors to give the appearance that the surface has bumps or indentations. How is this actually done? We will examine two methods.

Blinn (1978), the paper that introduced bump mapping, describes how a bump of height $F(u, v)$ along the normal vector N can be simulated. The modified or "perturbed" normal vector is $N' = N + D$. The perturbation vector D lies in the tangent

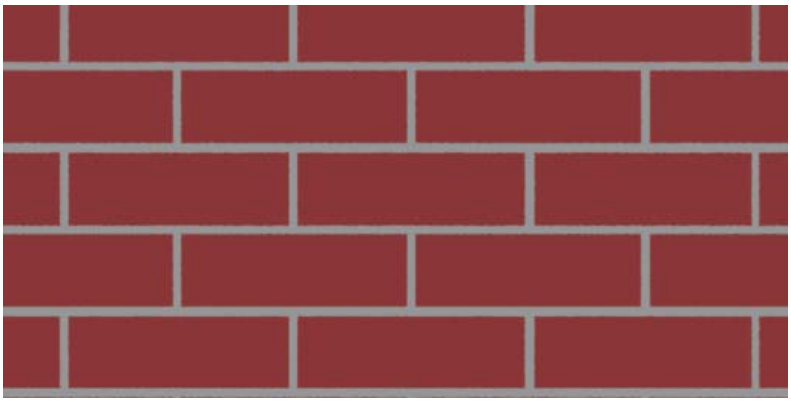


FIGURE 2.22 The brick texture.

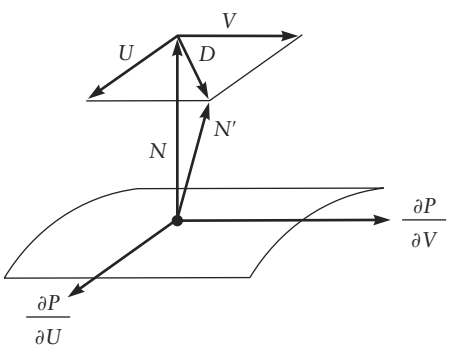


FIGURE 2.23 The geometry of bump mapping.

plane of the surface and is therefore perpendicular to N. D is based on the sum of two separate perturbation vectors U and V (Figure 2.23).

$$U = \frac{\partial F}{\partial u}\left(N \times \frac{\partial P}{\partial v}\right)$$

$$V = -\left(\frac{\partial F}{\partial v}\left(N \times \frac{\partial P}{\partial u}\right)\right)$$

$$D = \frac{1}{|N|}(U + V)$$

Team LRN

Let's analyze the expression for U. Note that the cross product

$$N \times \frac{\partial P}{\partial v}$$

is perpendicular to N and therefore lies in the tangent plane of the surface. It is also perpendicular to the partial derivative of P, the surface position, with respect to v. This derivative lies in the tangent plane and indicates the direction in which P changes as the surface parameter v is increased. If the parametric directions are perpendicular (usually they are only approximately perpendicular), adding a perturbation to N along the direction of this cross product would tilt N as if there were an upward slope in the surface along the u direction. The partial derivative $\partial F/\partial u$ gives the slope of the bump function in the u direction.

This technique looks somewhat frightening, but is fairly easy to implement if you already have the normal vector and the parametric derivatives of P. In the RenderMan shading language, `N`, `dPdu`, and `dPdv` contain these values. The differencing operators `Du` and `Dv` allow you to approximate the parametric derivatives of any expression. So Blinn's method of bump mapping could be implemented using the following shading language code:

```
float F; point U, V, D;
F = /* fill in some bump function here */
U = Du(F) * (N ^ dPdv);
V = -(Dv(F) * (N ^ dPdu));
D = 1/length(N) * (U + V);
Nf = N + D;
Nf = normalize(faceforward(Nf, I));
```

Then use `Nf` in the shading model just as you normally would. The resulting surface shading should give the appearance of a pattern of bumps determined by `F`.

Fortunately, the shading language provides a more easily remembered way to implement bump mapping and even displacement mapping.

```
float F;
point PP;

F = /* fill in some bump function here */

PP = P + F * normalize(N);
Nf = calculatenormal(PP);
Nf = normalize(faceforward(Nf, I));
```

In this code fragment, a new position `PP` is computed by moving along the direction of the normal a distance determined by the bump height `F`. Then the built-in

Team LRN

function `calculatenormal` is used to compute the normal vector of the modified surface `PP`. `calculatenormal(PP)` does nothing more than return the cross product of the parametric derivatives of the modified surface:

```
point
calculatenormal(point PP)
{
      return Du(PP) ^ Dv(PP);
}
```

To create actual geometric bumps by displacement mapping, you use very similar shading language code:

```
float F;

F = /* fill in some bump function here */

p = p + F * normalize(N);
N = calculatenormal(P);
```

Instead of creating a new variable `PP` that represents the bumped surface, this code assigns a new value to the original surface position `P`. In the shading language this means that the positions of points on the surface are actually moved by the shader to create bumps in the geometry. Similarly, the true normal vector `N` is recomputed so that it matches the displaced surface properly. We've omitted the last line that computes `Nf` because displacement mapping should be done in a separate displacement shader, not in the surface shader.[5]

To get a better understanding of bump mapping, let's add bump-mapped mortar grooves to our brick texture. The first step is to design the shape of the groove profile, that is, the vertical cross section of the bump function. Figure 2.24 is a diagram of the profile of the bricks and mortar grooves.

In order to realistically render the mortar groove between the bricks, we want the brick shader to compute a procedural bump-mapping function that will be used to adjust the normal vector before shading. To this end, we add the following code to the brick shader, immediately before the last statement (the one that computes `Ci` from the shading model).

5. Pixar's PhotoRealistic RenderMan renderer requires you to specify a *displacement bound* that tells the renderer what the maximum value of the bump height or other displacement will be. This is fully explained in the user's manual for the renderer.

Team LRN

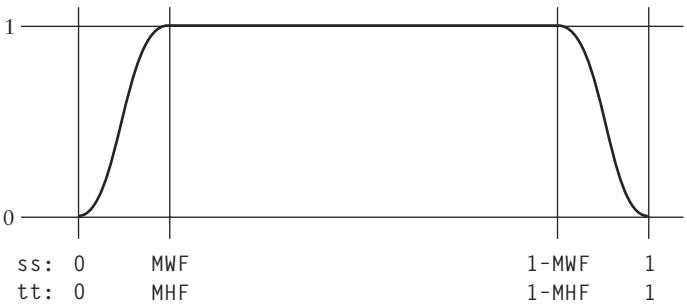


FIGURE 2.24 Brick and groove profile.

```
/* compute bump-mapping function for mortar grooves */
sbump = smoothstep(0,MWF,ss) - smoothstep(1-MWF,1,ss);
tbump = smoothstep(0,MHF,tt) - smoothstep(1-MHF,1,tt);
stbump = sbump * tbump;
```

The first two statements define the bump profile along the *s* and *t* directions independently. The first `smoothstep` call in each statement provides the positive slope of the bump function at the start of the brick, and the last `smoothstep` call in each statement provides the negative slope at the end of the brick. The last statement combines the `sbump` vertical groove and `tbump` horizontal groove to make an overall bump value `stbump`.

```
/* compute shading normal */
Nf = calculatenormal(P + normalize(N) * stbump);
Nf = normalize(faceforward(Nf, I));
Oi = Os;
Ci = Os * Ct * (Ka * ambient() + Kd * diffuse(Nf));
```

Finally, the shading normal `Nf` is computed based on the bump height as described earlier in this section. The shader ends as before by using the texture color `Ct` and bump-mapped normal `Nf` in a diffuse shading model to shade the surface. Figure 2.25 is an image of the bump-mapped brick texture.

There is a subtle issue hidden in this example. Recall that the shader displaces the surface position by a bump height `stbump` along the normal vector. Since the built-in normal vector `N` was used without modification, the displacement is defined in the shader's `current` space, not in `shader` space. Even though the bump function itself is locked to the surface because it is defined in terms of the `s` and `t` surface texture coordinates, the height of the bumps could change if the object is scaled relative

Team LRN

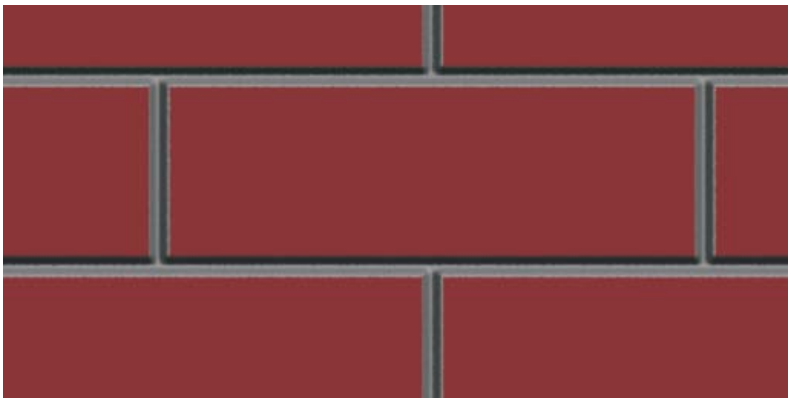


FIGURE 2.25 The bump-mapped brick texture.

to the `world` space. To avoid this problem, we could have transformed the surface point and normal vector into `shader` space, done the displacement there, and transformed the new normal back to `current` space, as follows:

```
point Nsh, Psh;
Psh = transform("shader", P);
Nsh = normalize(ntransform("shader", N));
Nsh = calculatenormal(Psh + Nsh * stbump);
Nf = ntransform("shader", "current", Nsh);
Nf = normalize(faceforward(Nf, I));
```

Note the use of `ntransform` rather than `transform` to transform normal vectors from one space to another. Normal vectors are transformed differently than points or direction vectors (see pages 216–217 of Foley et al. 1990). The second `ntransform` uses two space names to request a transformation from `shader` space to `current` space.

Example: Procedural Star Texture

Now let's try to generate a texture pattern that consists of a yellow five-pointed star on a background color `Cs`. The star pattern seems quite difficult until you think about it in polar coordinates. This is an example of how choosing the appropriate feature space makes it much easier to generate a tricky feature.

Figure 2.26 shows that each point of a five-pointed star is 72 degrees wide. Each half-point (36 degrees) is described by a single edge. The end points of the edge are a

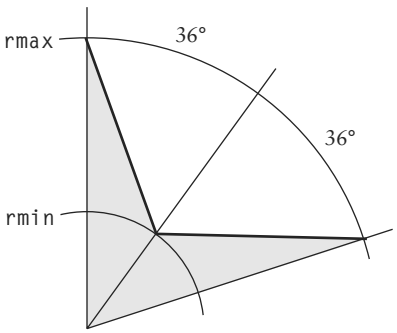


FIGURE 2.26 The geometry of a star.

point at radius `rmin` from the center of the star and another point at radius `rmax` from the center of the star.

```
surface
star(
      uniform float Ka = 1;
      uniform float Kd = 1;
      uniform color starcolor = color (1.0000,0.5161,0.0000);
      uniform float npoints = 5;
      uniform float sctr = 0.5;
      uniform float tctr = 0.5;
    )
{
      point Nf = normalize(faceforward(N, I));
      color Ct;
      float ss, tt, angle, r, a, in_out;
      uniform float rmin = 0.07, rmax = 0.2;
      uniform float starangle = 2*PI/npoints;
      uniform point p0 = rmax*(cos(0),sin(0), 0);
      uniform point pi = rmin*
          (cos(starangle/2),sin(starangle/2),0);
      uniform point d0 = pi - p0; point d1;
      ss = s - sctr; tt = t - tctr;
      angle = atan(ss, tt) + PI;
      r = sqrt(ss*ss + tt*tt);
```

At this point, the shader has computed polar coordinates relative to the center of the star. These coordinates `r` and `angle` act as the feature space for the star.

```
    a = mod(angle, starangle)/starangle;
    if (a >= 0.5)
          a = 1 - a;
```

Now the shader has computed the coordinates of the sample point `(r,a)` in a new feature space: the space of one point of the star. `a` is first set to range from 0 to 1 over each star point. To avoid checking both of the edges that define the "V" shape of the star point, sample points in the upper half of the star point are reflected through the center line of the star point. The new sample point `(r,a)` is inside the star if and only if the original sample point was inside the star, due to the symmetry of the star point around its center line.

```
    dl = r*(cos(a), sin(a),0) - p0;
    in_out = step(0, zcomp(d0^d1) );
    Ct = mix(Cs, starcolor, in_out);
    /* diffuse ("matte") shading model */
    Oi = Os;
    Ci = Os * Ct * (Ka * ambient() + Kd * diffuse(Nf));
}
```

To test whether `(r,a)` is inside the star, the shader finds the vectors `d0` from the tip of the star point to the `rmin` vertex and `d1` from the tip of the star point to the sample point. Now we use a handy trick from vector algebra. The cross product of two vectors is perpendicular to the plane containing the vectors, but there are two directions in which it could point. If the plane of the two vectors is the (x, y) plane, the cross product will point along the positive z-axis or along the negative z-axis. The direction in which it points is determined by whether the first vector is to the left or to the right of the second vector. So we can use the direction of the cross product to decide which side of the star edge `d0` the sample point is on.

Since the vectors `d0` and `d1` have z components of zero, the cross product will have x and y components of zero. Therefore, the shader can simply test the sign of `zcomp(d0^d1)`. We use `step(0,zcomp(d0^d1))` instead of `sign(zcomp(d0^d1))` because the `sign` function returns -1, 0, or 1. We want a binary (0 or 1) answer to the query "Is the sample point inside or outside the star?" This binary answer, `in_out`, is used to select the texture color `Ct` using the `mix` function, and the texture color is used to shade the sample point according to the diffuse shading model.

Figure 2.27 is an image rendered using the `star` shader.

Spectral Synthesis

Gardner (1984, 1985) demonstrated that procedural methods could generate remarkably complex and natural-looking textures simply by using a combination of sinusoidal component functions of differing frequencies, amplitudes, and phases. The theory of Fourier analysis tells us that functions can be represented as a sum

FIGURE 2.27 The star texture pattern.

of sinusoidal terms. The Fourier transform takes a function from the temporal or spatial domain, where it is usually defined, into the *frequency domain,* where it is represented by the amplitude and phase of a series of sinusoidal waves (Bracewell 1986; Brigham 1988). When the series of sinusoidal waves is summed together, it reproduces the original function; this is called the *inverse Fourier transform*.

Spectral synthesis is a rather inefficient implementation of the inverse discrete Fourier transform, which takes a function from the frequency domain back to the spatial domain. Given the amplitude and phase for each sinusoidal component, we can sum up the waves to get the desired function. The efficient way to do this is the inverse fast Fourier transform (FFT) algorithm, but that method generates the inverse Fourier transform for a large set of points all at once. In an implicit procedural texture we have to generate the inverse Fourier transform for a single sample point, and the best way to do that seems to be a direct summation of the sine wave components.

In procedural texture generation, we usually don't have all of the frequency domain information needed to reconstruct some function exactly. Instead, we want a function with some known characteristics, usually its power spectrum, and we don't care too much about the details of its behavior. It is possible to take a scanned image of a texture, compute its frequency domain representation using a fast Fourier transform, and use the results to determine coefficients for a spectral synthesis procedural texture, but in our experience that approach is rarely taken.

Team LRN

One of Gardner's simplest examples is a 2D texture that can be applied to a flat sky plane to simulate clouds. Here is a RenderMan shader that generates such a texture:

```
#define NTERMS 5
surface cloudplane(
    color cloudcolor = color (1,1,1);
    )
{
      color Ct;
      point Psh;
      float i, amplitude, f;
      float x, fx, xfreq, xphase;
      float y, fy, yfreq, yphase;
      uniform float offset = 0.5;
      uniform float xoffset = 13;
      uniform float yoffset = 96;

      Psh = transform("shader", P);
      x = xcomp(Psh) + xoffset;
      y = ycomp(Psh) + yoffset;

      xphase = 0.9; /* arbitrary */
      yphase = 0.7; /* arbitrary */
      xfreq = 2 * PI * 0.023;
      yfreq = 2 * PI * 0.021;
      amplitude = 0.3;
      f = 0;
      for (i = 0; i < NTERMS; i += 1) {
            fx = amplitude *
                  (offset + cos(xfreq * (x + xphase)));
            fy = amplitude *
                  (offset + cos(yfreq * (y + yphase)));
            f += fx * fy;
            xphase = PI/2 * 0.9 * cos (yfreq * y);
            yphase = PI/2 * 1.1 * cos (xfreq * x);

            xfreq *= 1.9 + i * 0.1; /* approximately 2 */
            yfreq *= 2.2 - i * 0.08; /* approximately 2 */
            amplitude *= 0.707;
      }
      f = clamp(f, 0, 1);

      Ct = mix(Cs, cloudcolor, f);
      Oi = Os;
      Ci = Os * Ct;
}
```

Team LRN

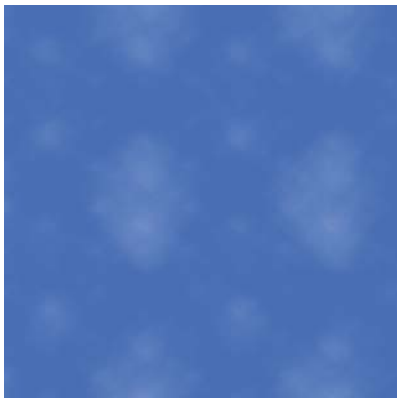


FIGURE 2.28 The cloud plane texture pattern.

This texture is a sum of five components, each of which is a cosine function with a different frequency, amplitude, and phase. The frequencies, amplitudes, and phases are chosen according to rules discovered by Gardner in his experiments. Gardner's technique is somewhat unusual for spectral synthesis in that the phase of each component is coupled to the value of the previous component in the other coordinate (for example, the x phase depends on the value of the preceding y component).

Making an acceptable cloud texture in this way is a battle to avoid regular patterns in the texture. Natural textures usually don't have periodic patterns that repeat exactly. Spectral synthesis relies on complexity to hide its underlying regularity and periodicity. There are several "magic numbers" strewn throughout this shader in an attempt to prevent regular patterns from appearing in the texture. Fourier spectral synthesis using a finite number of sine waves will always generate a periodic function, but the period can be made quite long so that the periodicity is not obvious to the observer. Figure 2.28 is an image rendered using the `cloudplane` shader with `Cs` set to a sky-blue color.

What Now?

You could go a long way using just the methods described so far. Some of these techniques can produce rich textures with a lot of varied detail, but even more variety is possible. In particular, we haven't yet discussed the noise function, the most popular of all procedural texture primitives. But first, let's digress a bit and examine one of the most important issues that affect procedural textures, namely, the difficulties of aliasing and antialiasing.

Team LRN

ALIASING AND HOW TO PREVENT IT

Aliasing is a term from the field of signal processing. In computer graphics, aliasing refers to a variety of image flaws and unpleasant artifacts that result from improper use of sampling. The staircaselike "jaggies" that can appear on slanted lines and edges are the most obvious examples of aliasing. The next section presents an informal discussion of basic signal processing concepts, including aliasing. For a more rigorous presentation, see Oppenheim and Schafer (1989), a standard signal processing textbook.

Signal Processing

As shown in Figure 2.29, a continuous signal can be converted into a discrete form by measuring its value at equally spaced sample points. This is called *sampling*. The original signal can be reconstructed later from the sample values by interpolation.

Sampling and reconstruction are fundamental to computer graphics.[6] Raster images are discrete digital representations of the continuous optical signals that nature delivers to our eyes and to our cameras. Synthetic images are made by sampling of geometric models that are mathematically continuous. Of course, our image signals are two-dimensional. Signal processing originally was developed to deal with the one-dimensional time-varying signals encountered in communications. The field of image processing is in essence the two-dimensional extension of signal processing techniques to deal with images.

Fortunately for computer graphics, the process of sampling and reconstruction is guaranteed to work under certain conditions, namely, when the amount of information in the original signal does not exceed the amount of information that can be captured by the samples. This is known as the *sampling theorem*. The amount of information in the original signal is called its *bandwidth*. The amount of information that can be captured by the samples is dependent upon the *sampling rate,* the number of sample points per unit distance. Unfortunately for computer graphics, the conditions for correct sampling and reconstruction are not always easy to meet, and when they are not met, aliasing occurs.

The theory of Fourier analysis tells us that a function can be represented as a sum of sinusoidal components with various frequencies, phases, and amplitudes. The Fourier transform converts the function from its original form in the "spatial

6. Mitchell and Netravali (1988) includes an excellent discussion of the many places in which sampling and reconstruction arise in computer graphics.

Team LRN

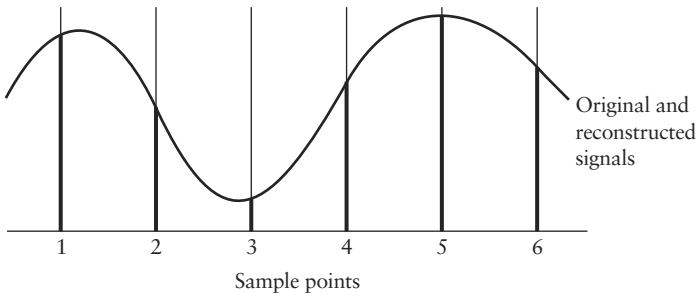


FIGURE 2.29 Sampling and reconstruction.

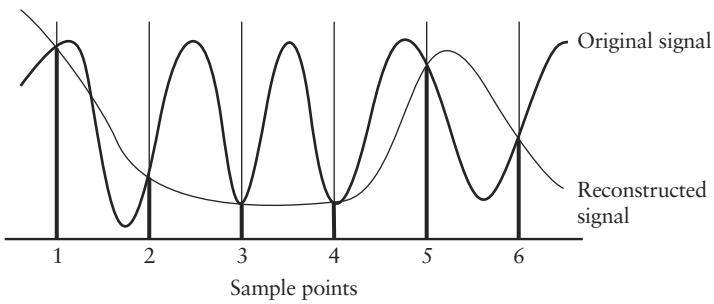


FIGURE 2.30 Aliasing.

domain" into a set of sinusoidal components in the "frequency domain." A signal with limited bandwidth will have a maximum frequency in its frequency domain representation. If that frequency is less than or equal to one-half of the sampling rate, the signal can be correctly sampled and reconstructed without aliasing. Aliasing will occur if the maximum frequency exceeds one-half of the sampling rate (this is called the *Nyquist frequency*). The maximum frequency in the reconstructed signal cannot exceed the Nyquist frequency, but the energy contributed to the original signal by the excessively high frequency components does not simply disappear. Instead, it appears in the reconstructed signal as erroneous lower-frequency energy, which is called an *alias* of the high-frequency energy in the original signal. Figure 2.30 illustrates this situation. The original signal varies too often to be adequately captured by the samples. Note that the signal reconstructed from the samples is quite different from the original signal. The problem of aliasing can be addressed by changing the sample points to be closer together, or by modifying the original signal to eliminate the high frequencies. If it is possible to increase the sampling rate, that is always beneficial. With more samples, the original signal can be reconstructed more accurately.

Team LRN

The Nyquist frequency threshold at which aliasing begins is increased, so the frequencies in the signal might now be below the Nyquist frequency.

Unfortunately, there is always some practical limit on the resolution of an image due to memory space or display limitations, and the sampling rate of an image is proportional to its resolution. It is impossible for an image to show details that are too small to be visible at the resolution of the image. Therefore, it is vital to take excessively high frequencies out of the original signal so that they don't show up as aliases and detract from the part of the signal that *can* be seen given the available resolution.

There is another reason why increasing the sampling rate is never a complete solution to the problem of aliasing. Some signals have unlimited bandwidth, so there is no maximum frequency. Sharp changes in the signal, for example, a `step` function, have frequency components of arbitrarily high frequency. No matter how great the image resolution, increasing the sampling rate to any finite value cannot eliminate aliasing when sampling such signals. This is why sloped lines are jaggy on even the highest resolution displays (unless they have been antialiased properly). Resolution increases alone can make the jaggies smaller, but never can eliminate them.

Since aliasing can't always be solved by increasing the sampling rate, we are forced to figure out how to remove high frequencies from the signal before sampling. The technique is called *low-pass filtering* because it passes low-frequency information while eliminating higher-frequency information.[7] The visual effect of low-pass filtering is to blur the image. The challenge is to blur the image as little as possible while adequately attenuating the unwanted high frequencies.

It is often difficult to low-pass-filter the signal before sampling. A common strategy in computer graphics is to *supersample* or *oversample* the signal, that is, to sample it at a higher rate than the desired output sampling rate. For example, we might choose to sample the signal four times for every output sample. If the signal were reconstructed from these samples, its maximum possible frequency would be four times the Nyquist frequency of the output sampling rate. A discrete low-pass filter can be applied to the "supersamples" to attenuate frequencies that exceed the Nyquist frequency of the output sampling rate. This method alleviates aliasing from frequencies between the output Nyquist frequency and the Nyquist frequency of the supersamples. Unfortunately, frequencies higher than the Nyquist frequency of the supersamples will still appear as aliases in the reconstructed signal.

7. In practice, effective antialiasing often requires a lower-frequency filtering criterion than the Nyquist frequency because the filtering is imperfect and the reconstruction of the signal from its samples is also imperfect.

Team LRN

An alternative approach to antialiasing is to supersample the signal at irregularly spaced points. This is called *stochastic sampling* (Cook, Porter, and Carpenter 1984; Cook 1986; Lee, Redner, and Uselton 1985; Dippé and Wold 1985). The energy from frequencies above the Nyquist frequency of the supersamples appears in the reconstructed signal as random noise rather than as a structured low-frequency alias. People are far less likely to notice this noise in the rendered image than they are to notice a low-frequency alias pattern. But it is preferable to low-pass-filter the signal before sampling because in that case no noise will be added to the reconstructed signal.

In summary, to produce an antialiased image with a specified resolution, the most effective strategy is to remove excessively high frequencies from the signal by low-pass filtering before sampling. If it isn't possible to filter the signal, the best strategy is to stochastically supersample it at as high a rate as is practical, and apply a discrete low-pass filter to the supersamples. The next section discusses ways to build low-pass filtering into procedural textures to eliminate aliasing artifacts.

You might wonder why aliasing is a problem in procedural textures. Doesn't the renderer do antialiasing? In fact, most renderers have some antialiasing scheme to prevent aliasing artifacts that result from sharp edges in the geometric model. Renderers that support image textures usually include some texture antialiasing in the texture mapping software. But these forms of antialiasing do not solve the aliasing problem for procedural textures.

The best case for "automatic" antialiasing of procedural textures is probably a stochastic ray tracer or, in fact, any renderer that stochastically supersamples the procedural texture. Rendering in this way is likely to be slow because of the many shading samples that are required by the supersampling process. And in the end, stochastic supersampling can only convert aliases into noise, not eliminate the unwanted high frequencies completely. If we can build a better form of antialiasing into the procedural texture, the result will look cleaner and the renderer can be freed of the need to compute expensive supersamples of the procedural texture.

PhotoRealistic RenderMan performs antialiasing by stochastically sampling the scene geometry and filtering the results of the sampling process (Cook, Carpenter, and Catmull 1987). The geometry is converted into a mesh of tiny quadrilaterals, and shading samples are computed at each vertex of the mesh before the stochastic sampling takes place. The vertices of the mesh are a set of samples of the location of the surface at equally spaced values of the surface parameters (u, v). Many shaders can be viewed as signal-generating functions defined on (u, v). A shader is evaluated at the mesh vertices, and the resulting colors and other properties are assigned to the mesh. This is really a sampling of the shader function at a grid of (u, v) values and its

Team LRN

reconstruction as a colored mesh of quadrilaterals. If the frequency content of the shader exceeds the Nyquist frequency of the mesh vertex (u, v) sampling rate, aliases will appear in the mesh colors. The reconstructed mesh color function is resampled by the stochastic sampling of the scene geometry. But once aliases have been introduced during the shader sampling process, they can never be removed by subsequent pixel sampling and filtering.

The separation of shader sampling from pixel sampling in PhotoRealistic RenderMan is advantageous because it permits a lower sampling rate for the shader samples than for the pixel samples. Shader samples are usually much more expensive to evaluate than pixel samples, so it makes sense to evaluate fewer of them. But this increases the need to perform some type of antialiasing in the shader itself; we can't rely on the stochastic supersampling of pixel samples to alleviate aliasing in procedural textures.

When image textures are used in the RenderMan shading language, the antialiasing is automatic. The texture system in the renderer filters the texture image pixels as necessary to attenuate frequencies higher than the Nyquist frequency of the (u, v) sampling rate in order to avoid aliasing.[8]

The brick texture from earlier in the chapter provides a concrete example of the aliasing problem. Figure 2.31 shows how the brick texture looks when the sampling rate is low. Notice that the width of the mortar grooves appears to vary in different parts of the image due to aliasing. This is the original version of the texture, without bump-mapped grooves. Later in the chapter we'll see how to add antialiasing techniques to the brick texture to alleviate the aliases.

Methods of Antialiasing Procedural Textures

By now you should be convinced that some form of antialiasing is necessary in procedural texture functions. In the remainder of this section we'll examine various ways to build low-pass filtering into procedural textures: clamping, analytic prefiltering, integrals, and alternative antialiasing methods. Clamping is a special-purpose filtering method that applies only to textures created by spectral synthesis. Analytic prefiltering techniques are ways to compute low-pass-filtered values for some of the primitive functions that are used to build procedural textures. One class of analytic prefiltering methods is based on the ability to compute the integral of the texture

8. See Feibush, Levoy, and Cook (1980), Williams (1983), Crow (1984), and Heckbert (1986a, 1986b) for detailed descriptions of methods for antialiasing image textures.

Team LRN

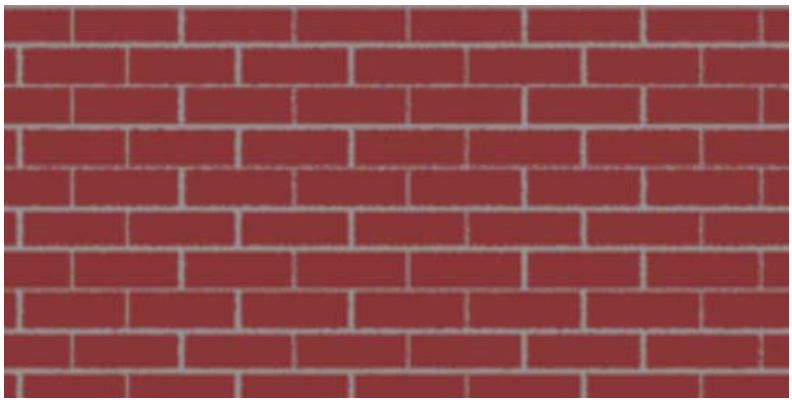


FIGURE 2.31 Aliasing in the brick texture.

function over a rectangular region. Finally, we'll consider alternatives to low-pass filtering that can be used when proper filtering is not practical.

Some procedural texture primitives are inherently band-limited; that is, they contain only a limited, bounded set of frequencies. `sin` is an obvious example of such a function. The `texture` function and its relatives have built-in filtering, so they are also band-limited. Unfortunately, some common language constructs such as `if` and `step` create sharp changes in value that generate arbitrarily high frequencies. Sharp changes in the shading function must be avoided. `smoothstep` is a smoothed replacement for `step` that can reduce the tendency to alias. Can we simply replace `step` functions with `smoothstep` functions?

The `smoothstep` function has less high-frequency energy than `step`, but using a particular `smoothstep` as a fixed replacement for `step` is not an adequate solution. If the shader is tuned for a particular view, the `smoothstep` will alias when the texture is viewed from further away because the fixed-width `smoothstep` will be too sharp. On the other hand, when the texture is viewed from close up, the `smoothstep` edge is too blurry. A properly antialiased edge should look equally sharp at all scales. To achieve this effect, the `smoothstep` width must be varied based on the sampling rate.

Determining the Sampling Rate

To do low-pass filtering properly, the procedural texture function must know the sampling rate at which the renderer is sampling the texture. The sampling rate is just the reciprocal of the spacing between adjacent samples in the relevant texture space

Team LRN

or feature space. This is called the *sampling interval*. For simple box filtering, the sampling interval is also the usual choice for the width of the box filter.

Obviously, the sampling interval cannot be determined from a single sample in isolation. Earlier parts of this chapter have presented a model of procedural texture in which the implicit texture function simply answers queries about the surface properties at a single sample point. The procedural texture is invoked many times by the renderer to evaluate the texture at different sample points, but each invocation is independent of all of the others.

To determine the sampling rate or sampling interval without changing this model of procedural texture, the renderer must provide some extra information to each invocation of the procedural texture. In the RenderMan shading language, this information is in the form of built-in variables called `du` and `dv` and functions called `Du` and `Dv`. The `du` and `dv` variables give the sampling intervals for the surface parameters (u, v). If the texture is written in terms of (u, v), the filter widths can be taken directly from `du` and `dv`.

In most cases, procedural textures are written in terms of the standard texture coordinates (s, t), which are scaled and translated versions of (u, v), or in terms of texture coordinates computed from the 3D coordinates of the surface point P in some space. In these cases, it is harder to determine the sampling interval, and the functions `Du` and `Dv` must be used. `Du(a)` gives an approximation to the derivative of some computed quantity `a` with respect to the surface parameter u. Similarly, `Dv(a)` gives an approximation to the derivative of some computed quantity `a` with respect to the surface parameter v. By multiplying the derivatives by the (u, v) sampling intervals, the procedural texture can estimate the sampling interval for a particular computed texture coordinate `a`. In general, it is not safe to assume that the texture coordinate changes only when u changes or only when v changes. Changes along both parametric directions have to be considered and combined to get a good estimate, `awidth`, of the sampling interval for `a`:

```
awidth = abs(Du(a)*du) + abs(Dv(a)*dv);
```

The sum of the absolute values gives an upper bound on the sampling interval; if this estimate is in error, it tends to make the filter too wide so that the result is blurred too much. This is safer than making the filter too narrow, which would allow aliasing to occur.

It is desirable for the sampling interval estimate to remain constant or change smoothly. Sudden changes in the sampling interval result in sudden changes in the texture filtering, and that can be a noticeable and annoying flaw in itself. Even if the

Team LRN

derivatives `Du` and `Dv` are accurate and change smoothly, there is no guarantee that the renderer's sampling intervals in (u, v) will also behave themselves. Many renderers use some form of adaptive sampling or adaptive subdivision to vary the rate of sampling depending on the apparent amount of detail in the image. In PhotoRealistic RenderMan, adaptive subdivision changes the shader sampling intervals depending on the size of the surface in the image. A surface seen in perspective could have sudden changes in sampling intervals between the nearer and more distant parts of the surface. A renderer that uses adaptive sampling based on some estimate of apparent detail might end up using the values returned by the procedural texture itself to determine the appropriate sampling rates. That would be an interesting situation indeed—one that might make proper low-pass filtering in the texture a very difficult task.

The remedy for cases in which the renderer's sampling interval is varying in an undesirable way is to use some other estimate of the sampling interval, an estimate that is both less accurate and smoother than the one described above. One such trick is to use the distance between the camera and the surface position to control the low-pass filtering:

```
awidth = length(I) * k;
```

The filter width (sampling interval estimate) is proportional to the distance from the camera `(length(I))`, but some experimentation is needed to get the right scaling factor `k`.

It is especially tricky to find the right filter width to antialias a bump height function for a bump-mapping texture. Since the bump height affects the normal vector used in shading, specular highlights can appear on the edges of bumps. Specular reflection functions have quite sharp angular falloff, and this sharpness can add additional high frequencies to the color output of the shader that are not in the bump height function. It might not be sufficient to filter the bump height function using the same low-pass filter that would be used for an ordinary texture that changes only the color or opacity. A wider filter probably is needed, but determining just how much wider it should be is a black art.

Clamping

Clamping (Norton, Rockwood, and Skolmoski 1982) is a very direct method of eliminating high frequencies from texture patterns that are generated by spectral synthesis. Since each frequency component is explicitly added to a spectral synthesis

Team LRN

texture, it is fairly easy to omit every component whose frequency is greater than the Nyquist frequency.

Let's begin with the following simple spectral synthesis loop, with a texture coordinate `s`:

```
value = 0;
for (f = MINFREQ; f < MAXFREQ; f *= 2)
      value += sin(2*PI*f*s)/f;
```

The loop begins at a frequency of `MINFREQ` and ends at a frequency less than `MAXFREQ`, doubling the frequency on each successive iteration of the loop. The amplitude of each sinusoidal component is the reciprocal of its frequency.

The following version is antialiased using the simplest form of clamping. The sampling interval in `s` is `swidth`.

```
value = 0;
cutoff = clamp(0.5/swidth, 0, MAXFREQ);
for (f = MINFREQ; f < cutoff; f *= 2)
      value += sin(2*PI*f*s)/f;
```

In this version the loop stops at a frequency less than `cutoff`, which is the Nyquist frequency for the sampling rate `1/swidth`. In order to avoid "pops," sudden changes in the texture as the sampling rate changes (e.g., as we zoom in toward the textured surface), it is important to fade out each component gradually as the Nyquist frequency approaches the component frequency. The following texture function incorporates this gradual fade-out strategy:

```
value = 0;
cutoff = clamp(0.5/swidth, 0, MAXFREQ);
for (f = MINFREQ; f < 0.5*cutoff; f *= 2)
      value += sin(2*PI*f*s)/f;
fade = clamp(2*(cutoff-f)/cutoff, 0, 1);
value += fade * sin(2*PI*f*s)/f;
```

The loop ends one component earlier than before, and that last component (whose frequency is between `0.5*cutoff` and `cutoff`) is added in after the loop and is scaled by `fade`. The `fade` value gradually drops from 1 to 0 as the frequency of the component increases from `0.5*cutoff` toward `cutoff`. This is really a result of changes in `swidth` and therefore in `cutoff`, rather than changes in the set of frequency components in the texture pattern.

Note that the time to generate the spectral synthesis texture pattern will increase as the sampling rate increases, that is, as we look more closely at the texture pattern. More and more iterations of the synthesis loop will be executed as the camera

Team LRN

approaches the textured surface. The example code incorporates `MAXFREQ` as a safety measure, but if `MAXFREQ` is reached, the texture will begin to look ragged when viewed even more closely.

Clamping works very well for spectral synthesis textures created with sine waves. It is hard to imagine a clearer and more effective implementation of low-pass filtering! But when the spectral synthesis uses some primitive that has a richer frequency spectrum of its own, clamping doesn't work as well.

If the primitive contains frequencies higher than its nominal frequency, the low-pass filtering will be imperfect and some high-frequency energy will leak into the texture. This can cause aliasing.

Even if the primitive is perfectly band-limited to frequencies lower than its nominal frequency, clamping is imperfect as a means of antialiasing. In this case, clamping will eliminate aliasing, but the character of the texture may change as high frequencies are removed because each component contains low frequencies that are removed along with the high frequencies.

Analytic Prefiltering

A procedural texture can be filtered explicitly by computing the convolution of the texture function with a filter function. This is difficult in general, but if we choose a simple filter, the technique can be implemented successfully. The simplest filter of all is the box filter; the value of a box filter is simply the average of the input function value over the area of the box filter.

To compute the convolution of a function with a box filter the function must be integrated over the area under the filter. This sounds tough, but it's easy if the function is simple enough. Consider the `step` function shown in Figure 2.8. The `step` function is rather ill-behaved because it is discontinuous at its threshold value. Let's apply a box filter extending from `x` to `x + w` to the function `step(b,x)`. The result is the box-filtered step function, `boxstep(a,b,x)`, where `a = b - w` (Figure 2.32). The value of `boxstep` is the area under the `step` function within the filter box. When the entire filter is left of `b` (that is, $x \geq b$), the value is 0. When the entire filter is right of `b` (that is, $x > b$), the value is 1. But `boxstep` is "smoother" than the `step` function; instead of being a sharp, discontinuous transition from 0 to 1 at `b`, `boxstep` is a linear ramp from 0 to 1 starting at `a` and ending at `b`. The slope of the ramp is `1/w`.

The `boxstep` function can be written as a preprocessor macro in C or the shading language as follows:

```
#define boxstep(a,b,x) clamp(((x)-(a))/((b)-(a)),0,1)
```

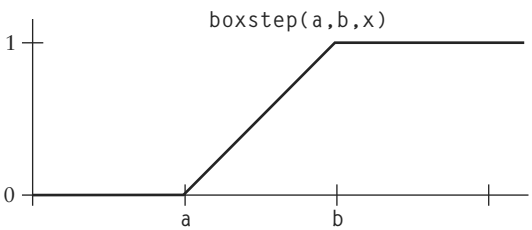


FIGURE 2.32 Box-filtering the `step` function.

Now the `step(b,x)` can be replaced with `boxstep(b-w,b,x)`. If the filter width `w` is chosen correctly, the `boxstep` function should reduce aliasing compared to the `step` function.

Better Filters

Now we know how to generate a box-filtered version of the `step` function, but the box filter is far from ideal for antialiasing. A better filter usually results in fewer artifacts or less unnecessary blurring. A better alternative to `boxstep` is the `smoothstep` function that was discussed earlier in this chapter. Filtering of the step with a first-order filter (box) gives a second-order function, namely, the linear ramp. Filtering of the step with a third-order filter (quadratic) gives a fourth-order function, namely, the cubic `smoothstep`. Using `smoothstep` to replace `step` is like filtering with a quadratic filter, which is a better approximation to the ideal sinc filter than the box filter is.

The `boxstep` macro is designed to be plug-compatible with `smoothstep`. The call `boxstep(WHERE-swidth, WHERE, s)` can be replaced with the call `smoothstep(WHERE-swidth, WHERE, s)`. This is the filtered version of `step(WHERE, s)`, given a filter extending from `s` to `s+swidth`.

Using the `smoothstep` cubic function as a filtered step is convenient and efficient because it is a standard part of the shading language. However, there are other filters and other filtered steps that are preferable in many applications. In particular, some filters such as the sinc and Catmull-Rom filters have *negative lobes*—the filter values dip below zero at some points. Such filters generally produce sharper texture patterns, although ringing artifacts are sometimes visible. A Catmull-Rom filter can be convolved with a step function (which is equivalent to integrating the filter function) to produce a `catstep` filtered step function that has been used with good results (Sayre 1992).

Team LRN

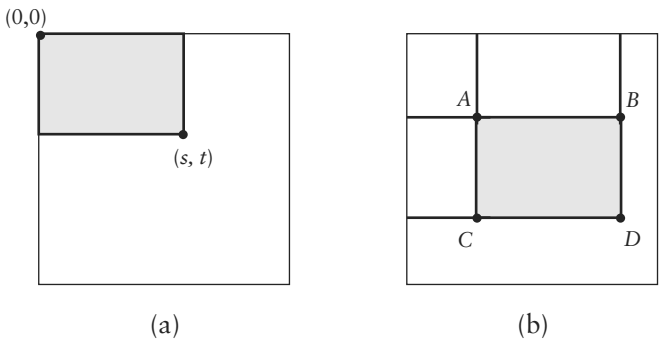


FIGURE 2.33 The summed-area table: (a) table entry (s, t) stores area of shaded region; (b) four entries A, B, C, D are used to compute shaded area.

Integrals and Summed-Area Tables

Crow (1984) introduced the *summed-area table* method of antialiasing image textures. A summed-area table is an image made from the texture image. As illustrated in Figure 2.33(a), the pixel value at coordinates (s, t) in the summed-area table is the sum of all of the pixels in the rectangular area $(0{:}s, 0{:}t)$ in the texture image. Of course, the summed-area table might need higher-precision pixel values than those of the original texture to store the sums accurately.

The summed-area table makes it easy to compute the sum of all of the texture image pixels in any axis-aligned rectangular region. Figure 2.33(b) shows how this is done. The pixel values at the corners of the region A, B, C, D are obtained from the summed-area table (four pixel accesses). The sum over the desired region is then simply $D + A - B - C$. This sum divided by the area of the region is the average value of the texture image over the region. If the region corresponds to the size and position of a box filter in the (s, t) space, the average value from the summed-area table calculation can be used as an antialiased texture value. The cost of the antialiasing is constant regardless of the size of the region covered by the filter, which is very desirable.

The summed-area table is really a table of the integral of the original texture image over various regions of the (s, t) space. An analogous antialiasing method for procedural textures is to compute the definite integral of the procedural texture over some range of texture coordinate values, rather than computing the texture value itself. For example, a procedural texture function $f(x)$ on some texture coordinate x might have a known indefinite integral function $F(x)$. If the desired box filter width is

Team LRN

w_x, the expression $(F(x) - F(x - w_x))/w_x$ might be used as a filtered alternative to the texture value $f(x)$. Integrals for many of the basic building blocks of procedural textures are easy to compute, but a few are tricky.[9]

Example: Antialiased Brick Texture

As an example of the application of these techniques, let's build antialiasing into the brick texture described earlier in this chapter.

The first step is to add the code needed to determine the filter width. The width variables must be added to the list of local variable declarations:

```
float swidth, twidth;
```

To compute the filter widths, we can add two lines of code just before the two lines that compute the brick numbers `sbrick` and `tbrick`:

```
swidth = abs(Du(ss)*du) + abs(Dv(ss)*dv);
twidth = abs(Du(tt)*du) + abs(Dv(tt)*dv);
sbrick = floor(ss); /* which brick? */
tbrick = floor(tt); /* which brick? */
```

The actual antialiasing is done by replacing the following two lines of the original shader that determine where to change from mortar color to brick color:

```
w = step(MWF,ss) - step(1-MWF,ss);
h = step(MHF,tt) - step(1-MHF,tt);
```

with an antialiased version of the code:

```
w = boxstep(MWF-swidth,MWF,ss)
  - boxstep(1-MWF-swidth,1-MWF,ss);
h = boxstep(MHF-twidth,MHF,tt)
  - boxstep(1-MHF-twidth,1-MHF,tt);
```

This is just the same code using `boxstep` instead of `step`. If the texture pattern consisted of a single brick in an infinite field of mortar, this would be sufficient. Unfortunately, more is required in order to handle a periodic pattern like the brick texture. The brick texture depends on a `mod`-like folding of the texture coordinates to convert a single pulse into a periodic sequence of pulses. But a wide filter positioned

9. The noise functions described in the later section "Making Noises" are among the tricky ones to integrate.

Team LRN

inside one brick can overlap another brick, a situation that is not properly accounted for in this periodic pulse scheme.

To solve the aliasing problem in a more general way, we can apply the integration technique described in the previous section. The integral of a sequence of square wave pulses is a function that consists of upward-sloping ramps and plateaus. The ramps correspond to the intervals where the pulses have a value of 1, and the plateaus correspond to the intervals where the pulses have a value of 0. In other words the slope of the integral is either 0 or 1, depending on the pulse value. The slope is the derivative of the integral, which is obviously the same as the original function.

The integrals of the periodic pulse functions in the `ss` and `tt` directions are given by the following preprocessor macros:

```
#define frac(x)        mod((x),1)
#define sintegral(ss)  (floor(ss)*(1-2*MWF) + \
                       max(0,frac(ss)-MWF))
#define tintegral(tt)  (floor(tt)*(1-2*MHF) + \
                       max(0,frac(tt)-MHF))
```

These are definite integrals from 0 to `ss` and 0 to `tt`. The `ss` integral consists of the integral of all of the preceding complete pulses (the term involving the floor function) plus the contribution of the current partial pulse (the term involving the fractional part of the coordinate).

To compute the antialiased value of the periodic pulse function, the shader must determine the value of the definite integral over the area of the filter. The value of the integral is divided by the area of the filter to get the average value of the periodic pulse function in the filtered region.

```
w = (sintegral(ss+swidth) - sintegral(ss))/swidth;
h = (tintegral(tt+twidth) - tintegral(tt))/twidth;
```

When using this method of antialiasing, you should remove the following lines of code from the shader:

```
ss -= sbrick;
tt -= tbrick;
```

because the `floor` and `mod` operations in the integrals provide the necessary periodicity for the pulse sequence. Forcing `ss` and `tt` to lie in the unit interval interferes with the calculation of the correct integral values.

Team LRN

Figure 2.34 shows the antialiased version of the brick texture, which should be compared with the original version shown in Figure 2.31. The widths of the mortar grooves are more consistent in the antialiased version of the texture.

Alternative Antialiasing Methods

Building low-pass filtering into a complicated procedural texture function can be far from easy. In some cases you might be forced to abandon the worthy goal of "proper" filtering and fall back on some alternative strategy that is more practical to implement.

One simple alternative to low-pass filtering is simply to blend between two or more versions of the texture based on some criterion related to sampling rate. For example, as the sampling rate indicates that the samples are getting close to the rate at which the texture begins to alias, you can fade your texture toward a color that is the average color of the texture. This is clearly a hack; the transition between the detailed texture and the average color might be quite obvious, although this is probably better than just letting the texture alias. The transition can be smoothed out by using more than two representations of the texture and blending between adjacent pairs of textures at the appropriate sampling rates.

A more sophisticated antialiasing method is to supersample the texture pattern in the procedural texture itself. When the shader is asked to supply the color of a sample point, it will generate several more closely spaced texture samples and combine them in some weighted sum that implements a low-pass filter. As mentioned earlier, supersampling will at least decrease the sampling rate at which aliasing

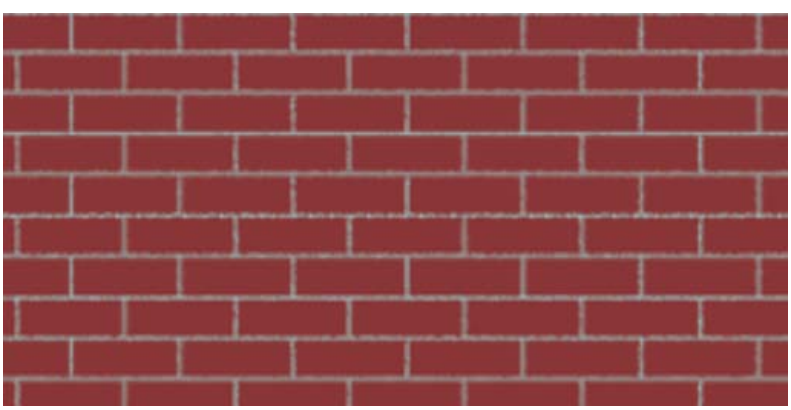


FIGURE 2.34 Box-filtered version of the brick texture.

Team LRN

begins. If the positions of the supersamples are jittered stochastically, the aliases will tend to be broken up into noise that might not be objectionable.

Supersampling in the procedural texture can be complicated and expensive, but it might be more efficient than supersampling implemented by the renderer. The procedural texture can limit the amount of code that is executed for each texture sample and therefore do a more lightweight version of supersampling.

MAKING NOISES

To generate irregular procedural textures, we need an irregular primitive function, usually called `noise`. This is a function that is apparently stochastic and will break up the monotony of patterns that would otherwise be too regular. When we use terms like "random" and "stochastic" in this discussion, we almost always mean to say "apparently random" or "pseudorandom." True randomness is unusual in computer science, and as you will see, it is actually undesirable in procedural textures.

We discussed the importance of aliasing and antialiasing before covering irregular patterns because issues related to antialiasing are of key importance in the design of stochastic texture primitives.

The obvious stochastic texture primitive is *white noise,* a source of random numbers, uniformly distributed with no correlation whatsoever between successive numbers. White noise can be generated by a random physical process, such as the thermal noise that occurs within many analog electronic systems. Try tuning a television to a channel on which no station is currently broadcasting if you want to see a good approximation to white noise.

A pseudorandom number generator produces a fair approximation to white noise. But is white noise really what we need? Alas, a bit of thought reveals that it is not. White noise is never the same twice. If we generate a texture pattern on the surface of some object, let's say a marble pattern, we certainly will want the pattern to stay the same frame after frame in an animation or when we look at the object from a variety of camera positions. In fact, we need a function that is apparently random but is a repeatable function of some inputs. Truly random functions don't have inputs. The desired stochastic texture primitive will take texture coordinates as its inputs and will always return the same value given the same texture coordinates.

Luckily, it isn't hard to design such a function. Looking into the literature of hashing and pseudorandom number generation, we can find several ways to convert a set of coordinate numbers into some hashed value that can be treated as a pseudorandom number (PRN). Alternatively, the hashed value can be used as an index into a table of previously generated PRNs.

Team LRN

Even this repeatable sort of white noise isn't quite what is needed in a stochastic texture primitive. The repeatable pseudorandom function has an unlimited amount of detail, which is another way of saying that its values at adjacent points are completely independent of one another (uncorrelated). This sounds like what we want, but it proves to be troublesome because of the prevalence of point sampling in computer graphics. If we view an object from a new camera angle, the positions of the sample points at which the texture function is evaluated will change. A good PRN function will change its value markedly if the inputs change even slightly. Consequently, the texture will change when the camera is moved, and we don't want that to happen.

Another way to look at this problem is in terms of aliasing. White noise has its energy spread equally over all frequencies, including frequencies much higher than the Nyquist frequency of the shading samples. The sampling rate can never be high enough to capture the details of the white noise.

To keep our procedural textures stable and to keep them from aliasing, we need a stochastic function that is smoother than white noise. The solution is to use a low-pass-filtered version of white noise.[10] In the remainder of this chapter, we refer to these filtered noise functions simply as `noise` functions.

The properties of an ideal `noise` function are as follows:

- `noise` is a repeatable pseudorandom function of its inputs.
- `noise` has a known range, namely, from -1 to 1.
- `noise` is band-limited, with a maximum frequency of about 1.
- `noise` doesn't exhibit obvious periodicities or regular patterns. Such pseudorandom functions are always periodic, but the period can be made very long and therefore the periodicity is not conspicuous.
- `noise` is *stationary*—that is, its statistical character should be translationally invariant.
- `noise` is *isotropic*—that is, its statistical character should be rotationally invariant.

The remainder of this section presents a number of implementations of noise that meet these criteria with varying degrees of success.

10. Low-pass-filtered noise is sometimes called *pink noise,* but that term is more properly applied to a stochastic function with a $1/f$ power spectrum.

Team LRN

Lattice Noises

Lattice noises are the most popular implementations of noise for procedural texture applications. They are simple and efficient and have been used with excellent results. Ken Perlin's *noise* function (Perlin 1985), "the function that launched a thousand textures," is a lattice noise of the gradient variety; an implementation equivalent to Perlin's is described on page 75.[11]

The generation of a lattice noise begins with one or more uniformly distributed PRNs at every point in the texture space whose coordinates are integers. These points form the *integer lattice*. The necessary low-pass filtering of the noise is accomplished by a smooth interpolation between the PRNs. To see why this works, recall that the correct reconstruction of a signal from a set of samples can never contain frequencies higher than the Nyquist frequency of the sample rate. Since the PRNs at the integer lattice points are equally spaced samples of white noise and since reconstruction from samples is a form of interpolation, it is reasonable to expect that the interpolated function will be approximately band-limited below the Nyquist frequency of the lattice interval. The quality of the resulting `noise` function depends on the nature of the interpolation scheme.

All lattice noises need some way to generate one or more pseudorandom numbers at every lattice point. The `noise` functions in this chapter use a table of PRNs that is generated the first time `noise` is called. To find the PRNs in the table that are to be used for a particular integer lattice point (`ix,iy,iz`), we'll use the following code:

```
#define TABSIZE          256
#define TABMASK          (TABSIZE-1)
#define PERM(x)          perm[(x)&TABMASK]
#define INDEX(ix,iy,iz) PERM((ix)+PERM((iy)+PERM(iz)))
```

The macro `INDEX` returns an index into an array with `TABSIZE` entries. The selected entry provides the PRNs needed for the lattice point. Note that `TABSIZE` must be a power of two so that performing `i&TABMASK` is equivalent to `i%TABSIZE`. As noted on page 31, using `i%TABSIZE` isn't safe, because it will yield a negative result if `i` is negative. Using the bitwise AND operation "&" avoids this problem.

The array `perm` contains a previously generated random permutation of the integers from zero to `TABMASK` onto themselves. Feeding sequential integers through the

11. In case you wish to compare the implementations, note that Ken describes his *noise* function in detail in Chapter 12.

Team LRN

permutation gives back a pseudorandom sequence. This hashing mechanism is used to break up the regular patterns that would result if `ix`, `iy`, and `iz` were simply added together to form an index into the `noiseTab` table. Here is a suitable `perm` array:

```
static unsigned char perm[TABSIZE] = {
225, 155, 210, 108, 175, 199, 221, 144, 203, 116,  70, 213,  69, 158,  33, 252,
  5,  82, 173, 133, 222, 139, 174,  27,   9,  71,  90, 246,  75, 130,  91, 191,
169, 138,   2, 151, 194, 235,  81,   7,  25, 113, 228, 159, 205, 253, 134, 142,
248,  65, 224, 217,  22, 121, 229,  63,  89, 103,  96, 104, 156,  17, 201, 129,
 36,   8, 165, 110, 237, 117, 231,  56, 132, 211, 152,  20, 181, 111, 239, 218,
170, 163,  51, 172, 157,  47,  80, 212, 176, 250,  87,  49,  99, 242, 136, 189,
162, 115,  44,  43, 124,  94, 150,  16, 141, 247,  32,  10, 198, 223, 255,  72,
 53, 131,  84,  57, 220, 197,  58,  50, 208,  11, 241,  28,   3, 192,  62, 202,
 18, 215, 153,  24,  76,  41,  15, 179,  39,  46,  55,   6, 128, 167,  23, 188,
106,  34, 187, 140, 164,  73, 112, 182, 244, 195, 227,  13,  35,  77, 196, 185,
 26, 200, 226, 119,  31, 123, 168, 125, 249,  68, 183, 230, 177, 135, 160, 180,
 12,   1, 243, 148, 102, 166,  38, 238, 251,  37, 240, 126,  64,  74, 161,  40,
184, 149, 171, 178, 101,  66,  29,  59, 146,  61, 254, 107,  42,  86, 154,   4,
236, 232, 120,  21, 233, 209,  45,  98, 193, 114,  78,  19, 206,  14, 118, 127,
 48,  79, 147,  85,  30, 207, 219,  54,  88, 234, 190, 122,  95,  67, 143, 109,
137, 214, 145,  93,  92, 100, 245,   0, 216, 186,  60,  83, 105,  97, 204,  52

};
```

This hashing technique is similar to the permutation used by Ken Perlin in his *noise* function.

Ward (1991) gives an implementation of a lattice noise in which the lattice PRNs are generated directly by a hashing function rather than by looking in a table of random values.

Value Noise

Given a PRN between −1 and 1 at each lattice point, a noise function can be computed by interpolating among these random values. This is called *value noise*. The following routine will initialize a table of PRNs for value noise:

```
#define RANDMASK OxVfffffff
#define RANDNBR ((random() & RANDMASK)/(double) RANDMASK)
```

Team LRN

```
float valueTab[TABSIZE];

void
valueTabInit(int seed)
{
      float *table = valueTab;
      int i;
      srandom(seed);
      for(i = 0; i < TABSIZE; i++)
      *table++ = 1. - 2.*RANDNBR;
}
```

Given this table, it is straightforward to generate the PRN for an integer lattice point with coordinates `ix`, `iy`, and `iz`:

```
float
vlattice(int ix, int iy, int iz)
{
      return valueTab[INDEX(ix,iy,iz)];
}
```

The key decision to be made in implementing value noise is how to interpolate among the lattice PRNs. Many different methods have been used, ranging from linear interpolation to a variety of cubic interpolation techniques. Linear interpolation is insufficient for a smooth-looking noise; value noise based on linear interpolation looks "boxy," with obvious lattice cell artifacts. The derivative of a linearly interpolated value is not continuous, and the sharp changes are obvious to the eye. It is better to use a cubic interpolation method so both the derivative and the second derivative are continuous. Here is a simple implementation using the cubic Catmull-Rom spline interpolation function shown on page 34:

```
float
vnoise(float x, float y, float z)
{
      int ix, iy, iz;
      int i, j, k;
      float fx, fy, fz;
      float xknots[4], yknots[4], zknots[4];
      static int initialized = 0;

      if (!initialized) {
         valueTabInit(665);
         initialized = 1;
         }
      ix = FLOOR(x);
      fx = x - ix;
```

Team LRN

```
    iy = FLOOR(y);
    fy = y - iy;

    iz = FLOOR(z);
    fz = z - iz;

    for (k = -1; k <= 2; k++) {
      for (j = -1; j <= 2; j++) {
        for (i = -1; i <= 2; i++)
           xknots[i+1] = vlattice(ix+i,iy+j,iz+k);
        yknots[j+1] = spline(fx, 4, xknots);
      }
      zknots[k+1] = spline(fy, 4, yknots);
    }
    return spline(fz, 4, zknots);
}
```

Since this is a cubic Catmull-Rom `spline` function in all three dimensions, the spline has 64 control points, which are the vertices of the 27 lattice cells surrounding the point in question. Obviously, this interpolation can be quite expensive. It might make sense to use a modified version of the `spline` function that is optimized for the special case of four knots and a parameter value that is known to be between 0 and 1.

A graph of a 1D sample of `vnoise` is shown in Figure 2.35(a), and an image of a 2D slice of the function is shown in Figure 2.36(a). Figure 2.37(a) shows its power spectrum. The noise obviously meets the criterion of being band-limited; it has no significant energy at frequencies above 1.

Many other interpolation schemes are possible for value noise. Quadratic and cubic B-splines are among the most popular. These splines don't actually interpolate the lattice PRN values; instead they approximate the values, which may lead to a narrower oscillation range (lower amplitude) for the B-spline noise. The lattice convolution noise discussed on page 78 can be considered a value noise in which the interpolation is done by convolving a filter kernel with the lattice PRN values.

Lewis (1989) describes the use of *Wiener interpolation* to interpolate lattice PRNs. Lewis claims that Wiener interpolation is efficient and provides a limited amount of control of the `noise` power spectrum.

Gradient Noise

Value noise is the simplest way to generate a low-pass-filtered stochastic function. A less obvious method is to generate a pseudorandom gradient vector at each lattice point and then use the gradients to generate the stochastic function. This is called

gradient noise. The *noise* function described by Perlin (1985) and Perlin and Hoffert (1989) was the first implementation of gradient noise. The RenderMan shading language `noise` function used in the irregular texture examples later in this chapter is a similar implementation of gradient noise.

The value of a gradient noise is 0 at all of the integer lattice points. The pseudorandom gradients determine its behavior between lattice points. The gradient method uses an interpolation based on the gradients at the eight corners of a single lattice cell, rather than the 64-vertex neighborhood used in the cubic interpolation method described in the previous section.

Our implementation of gradient noise begins by using the following routine to initialize the table of pseudorandom gradient vectors:

```
#include <math.h>

float gradientTab[TABSIZE*3];
```

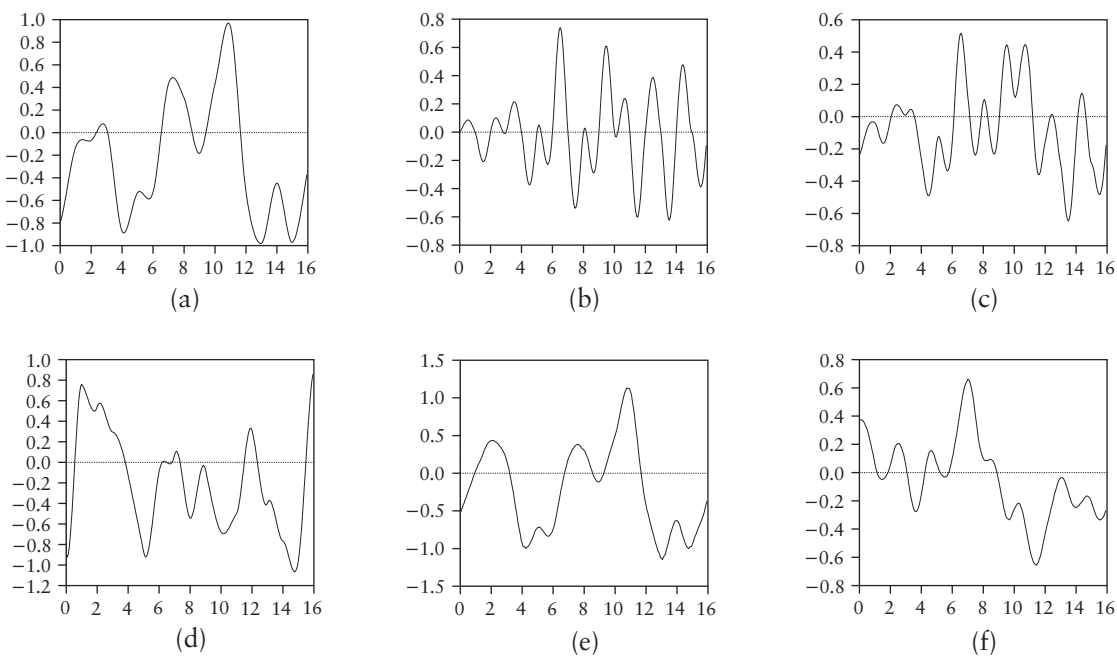


FIGURE 2.35 Graphs of various noises: (a) `vnoise`; (b) `gnoise` (Perlin's noise); (c) `vnoise + gnoise`; (d) Ward's Hermite noise; (e) `vcnoise`; (f) `scnoise`.

Team LRN

```
void
gradientTabInit(int seed)
{
      float *table = gradientTab;
      float z, r, theta;
      int i;

      srandom(seed);
      for(i = 0; i < TABSIZE; i++) {
            z = 1. - 2.*RANDNBR;
            /* r is radius of x,y circle */
            r = sqrtf(1 - z*z);
            /* theta is angle in (x,y) */
            theta = 2 * M_PI * RANDNBR;
```

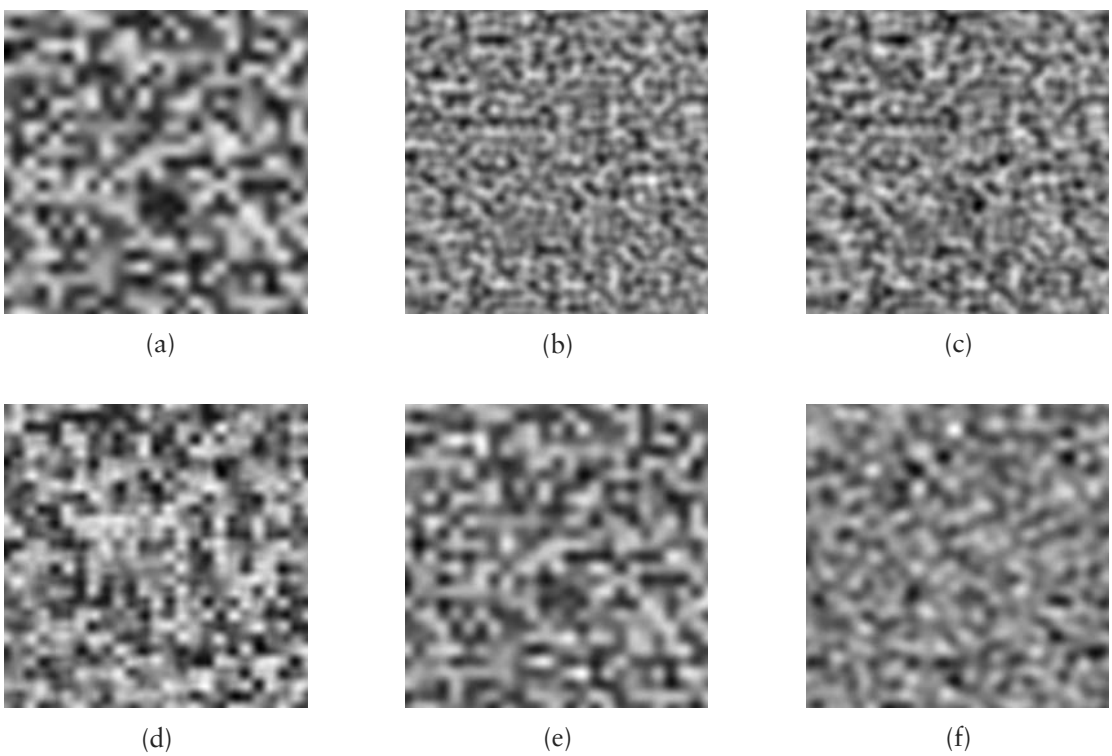


FIGURE 2.36 2D slices of various noises: (a) `vnoise`; (b) `gnoise` (Perlin's noise); (c) `vnoise` + `gnoise`; (d) Ward's Hermite noise; (e) `vcnoise`; (f) `scnoise`.

Team LRN

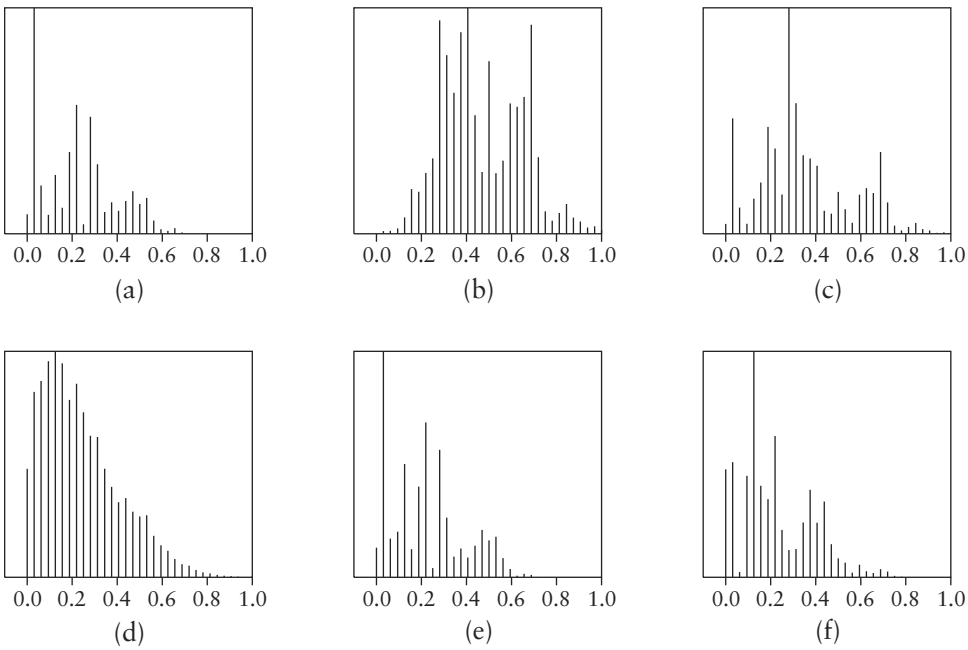


FIGURE 2.37 The power spectra of various noises: (a) `vnoise`; (b) `gnoise` (Perlin's noise); (c) `vnoise + gnoise`; (d) Ward's Hermite noise; (e) `vcnoise`; (f) `scnoise`.

```
            *table++ = r * cosf(theta);
            *table++ = r * sinf(theta);
            *table++ = z;
        }
}
```

This method of generating the gradient vectors attempts to produce unit vectors uniformly distributed over the unit sphere. It begins by generating a uniformly distributed `z` coordinate that is the sine of the latitude angle. The cosine of the same latitude angle is the radius `r` of the circle of constant latitude on the sphere. A second PRN is generated to give the longitude angle `theta` that determines the `x` and `y` components of the gradient.

Perlin's *noise* implementation uses a different scheme of generating uniformly distributed unit gradients. His method is to generate vectors with components between -1 and 1. Such vectors lie within the cube that bounds the unit sphere. Any vector whose length is greater than 1 lies outside the unit sphere and is discarded.

Team LRN

Keeping such vectors would bias the distribution in favor of the directions toward the corners of the cube. These directions have the greatest volume within the cube per solid angle. The remaining vectors are normalized to unit length.

The following routine generates the value of gradient noise for a single integer lattice point with coordinates `ix`, `iy`, and `iz`. The value `glattice` of the gradient noise for an individual lattice point is the dot product of the lattice gradient and the fractional part of the input point relative to the lattice point, given by `fx`, `fy`, and `fz`.

```
float
glattice(int ix, int iy, int iz,
float fx, float fy, float fz)
{
      float *g = &gradientTab[INDEX(ix,iy,iz)*3];
      return g[0]*fx + g[1]*fy + g[2]*fz;
}
```

Eight `glattice` values are combined using smoothed trilinear interpolation to get the gradient noise value. The linear interpolations are controlled by a `smoothstep`-like function of the fractional parts of the input coordinates.

```
#define LERP(t,x0,xl) ((x0) + (t)*((xl)-(x0)))
#define SMOOTHSTEP(x) ((x)*(x)*(3 - 2*(x)))

float
gnoise(float x, float y, float z)
{
      int ix, iy, iz;
      float fx0, fxl, fy0, fyl, fz0, fzl;
      float wx, wy, wz;
      float vx0, vxl, vy0, vyl, vz0, vzl;

      static int initialized = 0;

      if (!initialized) {
            gradientTabInit(665);
            initialized = 1;
      }
      ix = FLOOR(x);
      fx0 = x - ix;
      fxl = fx0 - 1;
      wx = SMOOTHSTEP(fx0);
      iy = FLOOR(y);
      fy0 = y - iy;
      fyl = fy0 - 1;
      wy = SMOOTHSTEP(fy0);
```

```
    iz = FLOOR(z);
    fz0 = z - iz;
    fzl = fz0 - 1;
    wz = SMOOTHSTEP(fz0) ;

    vx0 = glattice(ix,iy,iz,fx0,fy0,fz0);
    vxl = glattice(ix+l,iy,iz,fxl,fy0,fz0);
    vy0 = LERP(wx, vx0, vxl);
    vx0 = glattice(ix,iy+l,iz,fx0,fyl,fz0);
    vxl = glattice(ix+1,iy+1,iz,fxl,fyl,fz0);
    vyl = LERP(wx, vx0, vxl);
    vz0 = LERP(wy, vy0, vyl);
    vx0 = glattice(ix,iy,iz+l,fx0,fy0,fzl);
    vxl = glattice(ix+l,iy,iz+l,fxl,fy0,fzl);
    vy0 = LERP(wx, vx0, vxl);
    vx0 = glattice(ix,iy+l,iz+l,fx0,fyl,fzl);
    vxl = glattice(ix+l,iy+l,iz+l,fxl,fyl,fzl);
    vyl = LERP(wx, vx0, vxl);
    vzl = LERP(wy, vy0, vyl);

    return LERP(wz, vz0, vzl);
}
```

Figure 2.35(b) is a graph of a 1D sample of a gradient noise, and Figure 2.36(b) shows a 2D slice of the noise. Figure 2.37(b) shows its power spectrum. Most of the energy of gradient noise comes from frequencies between 0.3 and 0.7. There is more high-frequency energy in gradient noise than in value noise and less low-frequency energy. These are consequences of the fact that gradient noise has zeros at each lattice point and therefore is forced to change direction at least once per lattice step.

Value-Gradient Noise

A gradient noise is zero at all of the integer lattice points. This regular pattern of zeros sometimes results in a noticeable grid pattern in the gradient noise. To avoid this problem without losing the spectral advantages of gradient noise, we might try to combine the value and gradient methods to produce a value-gradient noise function.

One implementation of value-gradient noise is simple: it is just a weighted sum of a value noise and a gradient noise. Some computation can be saved by combining the two functions and sharing common code such as the `INDEX` calculation from the integer coordinates and the calculation of `ix, fx0,` and so on. Figure 2.35(c) shows a graph of a weighted sum of our Catmull-Rom `vnoise` and the gradient `gnoise,` and Figure 2.36(c) shows a 2D slice of it. Figure 2.37(c) shows the power spectrum

Team LRN

of this function. The slice image looks a little less regular than the gradient noise slice, presumably because the regular pattern of zero crossings has been eliminated.

A more sophisticated form of value-gradient noise is based on cubic Hermite interpolation. The Hermite spline is specified by its value and tangent at each of its end points. For a value-gradient noise, the tangents of the spline can be taken from the gradients. Ward (1991) gives the source code for just such a value-gradient noise. (Ward states that this is Perlin's *noise* function, but don't be fooled—it is quite different.) Figures 2.35(d) and 2.36(d) show Ward's Hermite noise function, and Figure 2.37(d) shows its power spectrum.[12] The power spectrum is remarkably regular, rising quite smoothly from DC to a frequency of about 0.2 and then falling smoothly down to 0 at a frequency of 1. Since the power is spread quite widely over the spectrum and the dominant frequencies are quite low, this noise function could be difficult to use in spectral synthesis.

Lattice Convolution Noise

One objection to the lattice noises is that they often exhibit axis-aligned artifacts. Many of the artifacts can be traced to the anisotropic nature of the interpolation schemes used to blend the lattice PRN values. What we'll call *lattice convolution noise* is an attempt to avoid anisotropy by using a discrete convolution technique to do the interpolation. The PRNs at the lattice points are treated as the values of random impulses and are convolved with a radially symmetrical filter. In the implementation that follows, we'll use a Catmull-Rom filter with negative lobes and a radius of 2. This means that any lattice point within a distance of two units from the input point must be considered in doing the convolution. The convolution is simply the sum of the product of each lattice point PRN value times the value of the filter function based on the distance of the input point from the lattice point.

Here is an implementation of lattice convolution noise called `vcnoise`. It begins with the filter function `catrom2`, which takes a squared distance as input to avoid the need to compute square roots. The first time `catrom2` is called, it computes a table of Catmull-Rom filter values as a function of squared distances. Subsequent calls simply look up values from this table.

12. This analysis of Ward's noise function is actually based on the source code that is provided on the diskette that accompanies *Graphics Gems IV*. The code on the diskette seems to be a newer version of Ward's routine with an improved interpolation method.

Team LRN

```
static float catrom2(float d)
{
#define SAMPRATE 100 /* table entries per unit distance */
#define NENTRIES (4*SAMPRATE+1)
      float x;
      int i;
      static float table[NENTRIES];
      static int initialized = 0;
      if (d >= 4)
            return 0;
      if (!initialized) {
            for (i = 0; i < NENTRIES; i++){
                  x = i/(float) SAMPRATE;
                  x = sqrtf(x);
                  if (x < 1)
                        table[i] = 0.5 * (2+x*x*(-5+x*3));
                  else
                        table[i] = 0.5 * (4+x*(-8+x*(5-x)));
            }
            initialized = 1;
      }
      d = d*SAMPRATE + 0.5;
      i = FLOOR(d);
      if (i >= NENTRIES)
            return 0;
      return table[i];
}

float
vcnoise(float x, float y, float z)
{
      int ix, iy, iz;
      int i, j, k;
      float fx, fy, fz;
      float dx, dy, dz;
      float sum = 0;
      static int initialized = 0;
      if (!initialized) {
            valueTabInit(665);
            initialized = 1;
      }
      ix = FLOOR(x);
      fx = x - ix;
      iy = FLOOR(y);
      fy = y - iy;
      iz = FLOOR(z);
      fz = z - iz;
```

```
        for (k = -1; k <= 2; k++) {
              dz = k - fz;
              dz = dz*dz;
              for (j = -1; j <= 2; j++) {
                    dy = j - fy;
                    dy = dy*dy;
                    for (i = -1; i <= 2; i++){
                          dx = i - fx;
                          dx = dx*dx;
                          sum += vlattice(ix+i,iy+j,iz+k)
                                * catrom2(dx + dy + dz);
                    }
              }
        }
        return sum;
}
```

Figure 2.35(e) shows a graph of `vcnoise`, and Figure 2.36(e) shows a 2D slice of it. Figure 2.37(e) shows its power spectrum. The spectrum is not unlike that of the other value noises. Perhaps this is not surprising since it is essentially a value noise with a different interpolation scheme. Some degree of spectral control should be possible by modifying the filter shape.

Sparse Convolution Noise

There are several ways to generate noise functions that aren't based on a regular lattice of PRNs. One such method is called *sparse convolution* (Lewis 1984, 1989). A similar technique called *spot noise* is described by van Wijk (1991).

Sparse convolution involves building up a noise function by convolving a filter function with a collection of randomly located random impulses (a Poisson process). The scattered random impulses are considered to be a "sparse" form of white noise, hence the term "sparse convolution." The low-pass filtering of the white noise is accomplished by the filter function. The power spectrum of the sparse convolution noise is derived from the power spectrum of the filter kernel, so some control of the noise spectrum is possible by modifying the filter.

Sparse convolution is essentially the same as the lattice convolution noise algorithm described in the previous section, except that the PRN impulse values are located at pseudorandom points in each lattice cell. Here is an implementation of `scnoise`, a sparse convolution noise based on Lewis's description. The filter used is the Catmull-Rom filter described in the previous section. Three randomly placed impulses are generated in each lattice cell. A neighborhood of 125 lattice cells must

Team LRN

be considered for each call to the noise function because a randomly placed impulse two cells away could have a nonzero filter value within the current cell. As a result this noise function is computationally expensive.

```
static float impulseTab[TABSIZE*4];
static void
impulseTabInit(int seed)
{
      int i;
      float *f = impulseTab;
      srandom(seed); /* Set random number generator seed. */
      for (i = 0; i < TABSIZE; i++) {
            *f++ = RANDNBR;
            *f++ = RANDNBR;
            *f++ = RANDNBR;
            *f++ = 1. - 2.*RANDNBR;
      }
}

#define NEXT(h) (((h)+1) & TABMASK)
#define NIMPULSES 3

float
scnoise(float x, float y, float z)
{
      static int initialized;
      float *fp;
      int i, j, k, h, n;
      int ix, iy, iz;
      float sum = 0;
      float fx, fy, fz, dx, dy, dz, distsq;

      /* Initialize the random impulse table if necessary. */
      if (!initialized) {
            impulseTabInit(665);
            initialized = 1;
      }
      ix = FLOOR (x); fx = x - ix;
      iy = FLOOR(y);  fy = y - iy;
      iz = FLOOR (z); fz = z - iz;

      /* Perform the sparse convolution. */
      for (i = -2; i <= 2; i++) {
            for (j = -2; j <= 2; j++) {
                  for (k = -2; k <= 2; k++) {
                        /* Compute voxel hash code. */
                        h = INDEX(ix+i,iy+j,iz+k);
```

```
                    for (n = NIMPULSES; n > 0; n--, h = NEXT(h)) {
                        /* Convolve filter and impulse. */
                        fp = &impulseTab[h*4];
                        dx = fx - (i + *fp++);
                        dy = fy - (j + *fp++);
                        dz = fz - (k + *fp++);
                        distsq = dx*dx + dy*dy + dz*dz;
                        sum += catrom2(distsq) * *fp;
                    }
                }
            }
        }

        return sum / NIMPULSES;
}
```

Figures 2.35(f) and 2.36(f) show 1D and 2D sections of `scnoise`. Figure 2.37(f) shows the power spectrum. The spectrum is similar to that of the other value noises, but the slice image appears to exhibit fewer gridlike patterns than the other noises.

Explicit Noise Algorithms

Some interesting methods of generating noises and random fractals aren't convenient for implicit procedural texture synthesis. These methods generate a large batch of noise values all at once in an explicit fashion. To use them in an implicit procedural texture during rendering, the noise values would have to be generated before rendering and stored in a table or texture image. A good example of such a technique is the midpoint displacement method (Fournier, Fussell, and Carpenter 1982). A related method of random successive additions is described by Saupe (1992). Lewis (1986, 1987) describes a generalization of such methods to give greater spectral control. Saupe (1989) shows that such methods can be more than an order of magnitude less expensive than implicit evaluation methods.

Fourier Spectral Synthesis

Another explicit method of noise generation is to generate a pseudorandom discrete frequency spectrum in which the power at a given frequency has a probability distribution that is correct for the desired noise. Then a discrete inverse Fourier transform (usually an inverse FFT) is performed on the frequency domain representation to get a spatial domain representation of the noise. Saupe (1988) and Voss (1988) describe this technique.

Team LRN

In the description of spectral synthesis textures on page 51, the example showed that many hand-picked "random" coefficients were used to generate the cloud texture. We could think of this as the generation of a random frequency domain representation and the evaluation of the corresponding spatial function using a spectral sum to implement the discrete inverse Fourier transform. This is far less efficient than an FFT algorithm, but has the advantage that it can be evaluated a point at a time for use in an implicit procedural texture. The complexity and apparent irregularity of Gardner's textures is less surprising when they are seen to be noiselike stochastic functions in disguise!

Direct Fourier synthesis of noise is much slower than the lattice noises described earlier and is probably not practical for procedural texture synthesis. Lattice convolution and sparse convolution are other methods that offer the promise of detailed spectral control of the noise. There is a trade-off between trying to generate all desired spectral characteristics in a single call to `noise` by using an expensive method such as Fourier synthesis or sparse convolution versus the strategy of building up spectral characteristics using a weighted sum of several cheaper gradient noise components.

Gradient noise seems to be a good primitive function to use for building up spectral sums of noise components, as demonstrated in the next section. When combining multiple `noise` calls to build up a more complex stochastic function, the gradient noise gives better control of the spectrum of the complex function because gradient noise has little low-frequency energy compared to the other noise functions; its dominant frequencies are near one-half.

GENERATING IRREGULAR PATTERNS

Armed with the stochastic primitive functions from the preceding section, we can now begin to generate irregular texture patterns. Since most natural materials are somewhat irregular and nonuniform, irregular texture patterns are valuable in simulating these materials. Even manufactured materials are usually irregular as a result of shipping damage, weathering, manufacturing errors, and so on. This section describes several ways to generate irregular patterns and gives examples in the form of RenderMan shaders.

The `noise` function in the RenderMan shading language is an implementation of the lattice gradient noise described in the preceding section. The RenderMan function is unusual in that it has been scaled and offset to range from 0 to 1, instead of the more usual range of -1 to 1. This means that the RenderMan `noise` function has a value of 0.5 at the integer lattice points. The -1 to 1 range is sometimes more

Team LRN

convenient, and we can use the following signed noise macro in RenderMan shaders to get a noise in this range:

```
#define snoise(x) (2 * noise(x) - 1)
```

The RenderMan `noise` function can be called with a 1D, 2D, or 3D input point, and will return a 1D or 3D result (a number, a point, or a color).

Most textures need several calls to `noise` to independently determine a variety of stochastic properties of the material. Remember that repeated calls to `noise` with the same inputs will give the same results. Different results can be obtained by shifting to another position in the noise space. It is common to call

```
noise(Q * frequency + offset)
```

where `offset` is of the same type as the coordinate `Q` and has the effect of establishing a new noise space with a different origin point.

There are two approaches to generating time-dependent textures. Textures that move or flow can be produced by moving through the 3D noise space over time:

```
f = noise(P - time*D);
```

can be used to make the texture appear to move in the direction and rate given by the vector `D` as time advances. Textures that simply evolve without a clear flow direction are harder to create. A 4D `noise` function is the best approach. RenderMan's `noise` function is limited to three dimensions.[13] In some cases one or two dimensions of the `noise` result can be used to represent spatial position, so that the remaining dimension can be used to represent time.

The shading language function `pnoise` is a relative of `noise`. The noise space can be wrapped back on itself to achieve periodic effects. For example, if a period of 30 is specified, `pnoise` will give the same value for input $x + 30$ as for input x. The oscillation frequency of the noise value is unaffected. Here are some typical `pnoise` calls:

```
pnoise(f, 30 );
pnoise(s, t, 30, 30)
pnoise(P, point (10, 15, 30))
```

13. We use a 4D quadratic B-spline value noise in our in-house animation system with good results. It pays to keep the order of the interpolation fairly low for 4D noise to avoid having too many lattice point terms in the interpolation.

Team LRN

It is easy to implement `pnoise` by making the choice of lattice PRNs periodic with the desired period. This technique is limited to integer periods.

When generating procedural textures using any of the `noise` functions described in this chapter, it is important to develop an understanding of the range and distribution of the noise values. It can be difficult to normalize a stochastic function so that the range is exactly −1 to 1. Furthermore, most of the noise values tend to lie close to 0 with only occasional excursions to the limits of the range. A histogram of the distribution of the noise values is a useful tool in designing functions based on `noise`.

Spectral Synthesis

The discussion on page 48 showed how complex regular functions with arbitrary spectral content can be built up from sine waves. As mentioned earlier, the many pseudorandom coefficients in Gardner's spectral synthesis textures might be viewed as a way of generating a noise function by the inverse Fourier transform method. Spectral synthesis using a noise function as the primitive gives an even richer stochastic content to the texture and reduces the need to use random coefficients for each component. When a stochastic function with a particular power spectrum is needed, spectral synthesis based on `noise` can be used to generate it.

Several calls to `noise` can be combined to build up a stochastic spectral function with a particular frequency/power spectrum. A noise loop of the form

```
value = 0;
for (f = MINFREQ; f < MAXFREQ; f *= 2)
      value += amplitude * snoise(Q * f);
```

with amplitude varying as a function of frequency `f` will build up a value with a desired spectrum. `Q` is the sample point in some texture space.

Perlin's well-known `turbulence` function is essentially a stochastic function of this type with a "fractal" power spectrum, that is, a power spectrum in which amplitude is proportional to `1/f`.

```
float
fractalsum(point Q)
{
      float value = 0;
      for (f = MINFREQ; f < MAXFREQ; f *= 2)
        value += snoise(Q * f)/f;
      return value;
}
```

Team LRN

This isn't quite the same as `turbulence`, however. Derivative discontinuities are added to the `turbulence` function by using the absolute value of the `snoise` function. Taking the absolute value folds the function at each zero crossing, making the function undifferentiable at these points. The number of peaks in the function is doubled, since the troughs become peaks.

```
float
turbulence(point Q)
{
      float value = 0;
      for (f = MINFREQ; f < MAXFREQ; f *= 2)
        value += abs(snoise(Q * f))/f;
      return value;
}
```

Figure 2.38 shows a slice of the `fractalsum` function on the left and a slice of the `turbulence` function on the right. The `fractalsum` is very cloudlike in appearance, while `turbulence` is apparently lumpier, with sharper changes in value. Figure 2.39(a) shows the power spectrum of the `fractalsum` function, and Figure 2.39(b) shows the power spectrum of the `turbulence` function. As you might expect, the power spectra show a rapid decline in energy as the frequency increases; this is a direct result of the $1/f$ amplitude scaling in the spectral synthesis loops.

Spectral synthesis loops should use clamping to prevent aliasing. Here is a version of `turbulence` with clamping included:

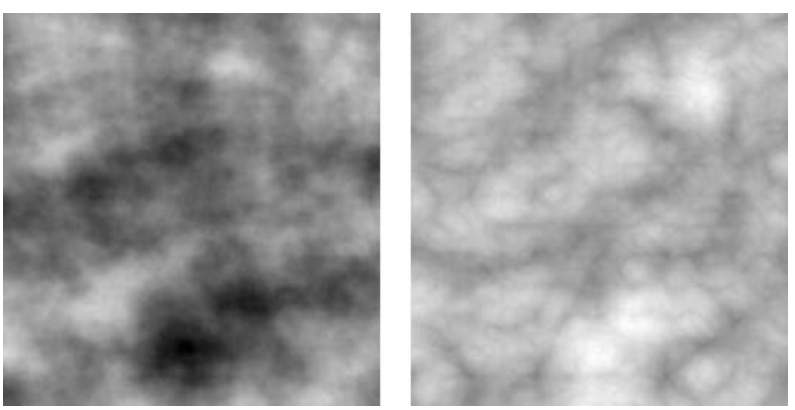


FIGURE 2.38 Slices of `fractalsum` and `turbulence` functions.

Team LRN

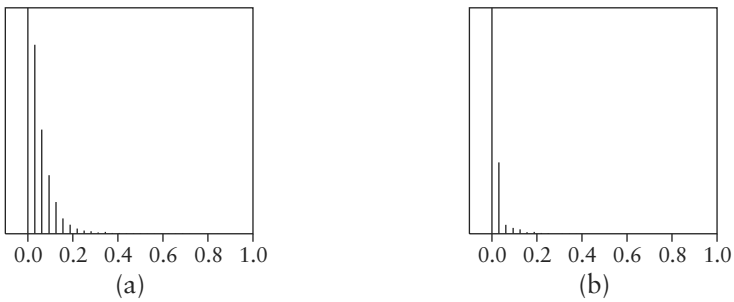


FIGURE 2.39 Power spectra of (a) `fractalsum` and (b) `turbulence` functions.

```
float
turbulence(point Q)
{
      float value = 0;
      float cutoff = clamp(0.5/Qwidth, 0, MAXFREQ);
      float fade;


      for (f = MINFREQ; f < 0.5*cutoff; f *= 2)
        value += abs(snoise(Q * f))/f;
      fade = clamp(2*(cutoff-f)/cutoff, 0, 1);
      value += fade * abs(snoise(Q * f))/f;
      return value;
}
```

Marble is a material that is typically simulated using an irregular texture based on spectral synthesis. The following marble shader uses a four-octave spectral synthesis based on `noise` to build up a stochastic value called `marble` that is similar to `fractalsum`. It is best to use a solid texture space for a marble texture, so that the texture can be used to shade curved surfaces as if they were carved out of a solid block of marble. This is accomplished by using the 3D surface point as the argument to the `noise` calls.

```
#define PALE_BLUE    color (0.25, 0.25, 0.35)
#define MEDIUM_BLUE  color (0.10, 0.10, 0.30)
#define DARK_BLUE    color (0.05, 0.05, 0.26)
#define DARKER_BLUE  color (0.03, 0.03, 0.20)
#define NNOISE       4
```

Team LRN

```
color
marble_color(float m)
{
      return color spline(
            clamp(2 * m + .75, 0, 1),
            PALE_BLUE, PALE_BLUE,
            MEDIUM_BLUE, MEDIUM_BLUE, MEDIUM_BLUE,
            PALE_BLUE, PALE_BLUE,
            DARK_BLUE, DARK_BLUE,
            DARKER_BLUE, DARKER_BLUE,
            PALE_BLUE, DARKER_BLUE);
}


surface blue_marble(
      uniform float Ka = 1;
      uniform float Kd = 0.8;
      uniform float Ks = 0.2;
      uniform float texturescale = 2.5;
      uniform float roughness = 0.1;
    )
{
      color Ct;
      point NN;
      point PP;
      float i, f, marble;


      NN = normalize(faceforward(N, I));
      PP = transform("shader", P) * texturescale;
      marble = 0; f = 1;
      for (i = 0; i < NNOISE; i += 1) {
            marble += snoise(PP * f)/f;
            f *= 2.17;
      }
      Ct = marble_color(marble);
      Ci = Os * (Ct * (Ka * ambient() + Kd * diffuse(NN))
            + Ks * specular(NN, normalize(-I), roughness));
}
```

The function `marble_color` maps the floating-point number `marble` into a color using a color spline. Figure 2.40 shows an example of the marble texture.

The spectral synthesis loop in this shader has no clamping control to avoid aliasing. If the texture is viewed from far away, aliasing artifacts will appear.

Note that the frequency multiplier in the spectral synthesis loop is 2.17 instead of exactly 2. This is usually better because it prevents the alignment of the lattice points of successive `snoise` components, and that tends to reduce lattice artifacts. In

Team LRN

FIGURE 2.40 Blue marble texture.

Chapter 6, Steve Worley describes another way to reduce lattice artifacts: randomly rotating the texture spaces of the different components.

Changing the frequency multiplier from 2 to some other value (like 2.17) affects the average size of gaps or *lacunae* in the texture pattern. In the fractal literature, this property of the texture is called *lacunarity* (Mandelbrot 1982). The scaling of the amplitude of successive components affects the *fractal dimension* of the texture. The $1/f$ amplitude scaling used here gives a $1/f^2$ scaling of the power spectrum, since power spectral density is proportional to amplitude squared. This results in an approximation to fractional Brownian motion (fBm) (Saupe 1992).

Perturbed Regular Patterns

Purely stochastic patterns tend to have an amorphous character. Often the goal is a more structured pattern with some appearance of regularity. The usual approach is to start with a regular pattern and add noise to it in order to make it look more interesting and more natural.

For example, the brick texture described on page 39 is unrealistically regular, as if the bricklayer were inhumanly precise. To add some variety to the texture, `noise` can be used to modify the relative positions of the bricks in the different rows. The following is the code from the original shader that calculates which brick contains the sample point.

```
sbrick = floor(ss); /* which brick? */
tbrick = floor(tt); /* which brick? */
ss -= sbrick;
tt -= tbrick;
```

Team LRN

To perturb the `ss` location of the bricks, we can rewrite this code as follows:

```
tbrick = floor(tt); /* which brick? */
ss += 0.1 * snoise(tbrick+0.5);
sbrick = floor(ss); /* which brick? */
ss -= sbrick;
tt -= tbrick;
```

There are a few subtle points to note here. The call to `snoise` uses `tbrick` rather than `tt` so that the noise value is constant over the entire brick. Otherwise, the stochastic offset would vary over the height of the brick and make the vertical edges of the brick wavy. Of course, we had to reorder the calculation of `sbrick` and `tbrick` so that `sbrick` can depend on `tbrick`. The value of the perturbation will change *suddenly* as `tbrick` jumps from one integer to the next. That's okay in this case, because the perturbation affects only the horizontal position of a row of bricks, and changes only in the middle of the mortar groove between the rows.

Since `snoise` is a gradient noise that is zero at all integer points, the perturbation always would be zero if the shader used `snoise(tbrick)`. Instead, it uses `snoise(tbrick + 0.5)` to sample the value of the noise halfway between the integer points, where it should have an interesting value.

The 0.1 multiplier on the `snoise` simply controls the size of the irregularity added to the texture. It can be adjusted as desired.

A more realistic version of this brick texture would incorporate some noise-based variations in the color of the bricks and mortar as well as a small stochastic bump mapping of the normal to simulate roughness of the bricks. It is easy to keep layering more stochastic effects onto a texture pattern to increase its realism or visual appeal. But it is important to begin with a simple version of the basic pattern and get that texture working reliably before attempting to add details and irregularity.

Perturbed Image Textures

Another valuable trick is to use a stochastic function to modify the texture coordinates used to access an image texture. This is very easy to do. For example, the simple texture access

```
Ct = texture("example.tx", s, t);
```

using the built-in texture coordinates `s` and `t` can be replaced with the following:

Team LRN

```
point Psh;
float ss, tt;
Psh = transform("shader", P);
ss = s + 0.2 * snoise(Psh);
tt = t + 0.2 * snoise(Psh+(1.5,6.7,3.4));
Ct = texture("example.tx", ss, tt);
```

In this example, `snoise` based on the 3D surface position in "shader" space is used to modify the texture coordinates slightly. Figure 2.41(a) shows the original image texture, and Figure 2.41(b) is a texture produced by perturbing the image texture with the code above.

Random Placement Patterns

A random placement pattern is a texture pattern that consists of a number of regular or irregular subpatterns or "bombs" that are dropped in random positions and orientations to form the texture. This bombing technique (Schacter and Ahuja 1979) was originally used in an explicit form to generate image textures before rendering. With some difficulty, it can be used in an implicit texture function during rendering.

The most obvious implementation is to store the positions of bombs in a table and to search the table for each sample point. This is rather inefficient and is especially hard to implement in the RenderMan shading language since the language has no tables or arrays. With a little ingenuity, we can devise a method of bombing that uses only `noise` to determine the bomb positions relevant to a sample point. The

(a) (b)

FIGURE 2.41 Perturbing an image texture: (a) original image; (b) perturbed image.

Team LRN

texture space is divided into a grid of square cells, with a bomb located at a random position within each cell.

In the following example shader, the bomb is the star pattern created by the procedural texture on page 46.

```
#define NCELLS 10
#define CELLSIZE (1/NCELLS)
#define snoise(s,t) (2*noise((s),(t))-1)

surface wallpaper(
      uniform float Ka = 1;
      uniform float Kd = 1;
      uniform color starcolor = color (1.0000,0.5161,0.0000);
      uniform float npoints = 5;
      )
{
      color Ct;
      point Nf;
      float ss, tt, angle, r, a, in_out;
      float sctr, tctr, scell, tcell;
      uniform float rmin = 0.01, rmax = 0.03;
      uniform float starangle = 2*PI/npoints;
      uniform point p0 = rmax*(cos(0),sin(0), 0);
      uniform point pi = rmin*
            (cos(starangle/2),sin(starangle/2), 0);
      uniform point d0 = pi - p0;
      point d1;

      scell = floor(s*NCELLS);
      tcell = floor(t*NCELLS);
      sctr = CELLSIZE * (scell + 0.5
         + 0.6 * snoise(scell+0.5, tcell+0.5));
      tctr = CELLSIZE * (tcell + 0.5
         + 0.6 * snoise(scell+3.5, tcell+8.5));
      ss = s - sctr;
      tt = t - tctr;
      angle = atan(ss, tt) + PI;
      r = sqrt(ss*ss + tt*tt);
      a = mod(angle, starangle)/starangle;

      if (a >= 0.5)
            a = 1 - a;
      d1 = r*(cos(a), sin(a),0) - p0;
      in_out = step(0, zcomp(d0^d1) );
      Ct = mix(Cs, starcolor, in_out);

      /* "matte" reflection model */
      Nf = normalize(faceforward(N, I));
```

Team LRN

```
      Oi = Os;
      Ci = Os * Ct * (Ka * ambient() + Kd * diffuse(Nf));
}
```

A couple of improvements can be made to this procedural texture. The requirement to have exactly one star in each cell makes the pattern quite regular. A separate noise value for each cell could be tested to see whether the cell should contain a star.

If a star is so far from the center of a cell that it protrudes outside the cell, this shader will clip off the part of the star that is outside the cell. This is a consequence of the fact that a given star is "noticed" only by sample points that lie inside the star's cell. The shader can be modified to perform the same shading tests for the eight cells that surround the cell containing the sample point. Stars that crossed over the cell edge will then be rendered correctly. The tests can be done by a pair of nested for loops that iterate over −1, 0, and 1. The nested loops generate nine different values for the cell coordinate pair `(scell, tcell)`. The star in each cell is tested against the sample point.

```
scellctr = floor(s*NCELLS);
tcellctr = floor(t*NCELLS);
in_out = 0;
for (i = -1; i <= 1; i += 1) {
      for (j = -1; j <= 1; j += 1) {
      scell = scellctr + i;
      tcell = tcellctr + j;
      if (noise(3*scell-9.5,7*tcell+7.5) < 0.55) {
            sctr = CELLSIZE * (scell + 0.5
                  + 0.6 * snoise(scell+0.5, tcell+0.5));
            tctr = CELLSIZE * (tcell + 0.5
                  + 0.6 * snoise(scell+3.5, tcell+8.5));
            ss = s - sctr;
            tt = t - tctr;

            angle = atan(ss, tt) + PI;
            r = sqrt(ss*ss + tt*tt);
            a = mod(angle, starangle)/starangle;

            if (a >= 0.5)
                  a = 1 - a;
            d1 = r*(cos(a), sin(a),0) - p0;
            in_out += step(0, zcomp(d0^d1));
      }
   }
}
Ct = mix(Cs, starcolor, step(0.5, in_out));
```

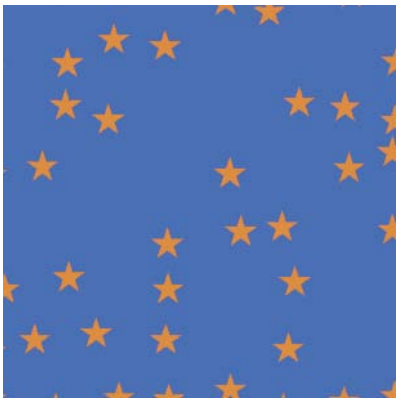


FIGURE 2.42 Random placement wallpaper texture.

The first `noise` call is used to decide whether or not to put a star in the cell. Note that the value of `in_out` can now be as high as 9. The additional `step` call in the last line converts it back to a 0 to 1 range so that the `mix` will work properly. Of course, the `step` functions in the wallpaper shader should be `smoothstep` calls with appropriate filter widths to prevent jaggy edges on the stars. Figure 2.42 shows the star wallpaper pattern generated by the shader.

The wallpaper texture in Pixar's film *Knickknack* (Figure 2.2) is a more elaborate example of a procedural texture based on bombing.

Note that the bomb subpattern used for a random placement texture doesn't have to be a procedural texture like the star. An image texture can be used instead, so that the bombing procedure can lay down copies of any picture you like, instead of the stars.

CONCLUSION

This chapter describes an approach to constructing procedural textures and a set of building blocks that you can use to build them. The range of textures that you can build is limited only by your imagination and your programming expertise. We hope that the examples in this chapter and in the rest of the book will inspire you and enable you to create wonderful imagery of your own, using procedural textures and models.

Team LRN

3

REAL-TIME PROGRAMMABLE SHADING

WILLIAM R. MARK

INTRODUCTION

The materials and lighting effects in the real world are very complex, but for many years real-time graphics hardware could only support a few simple shading models. VLSI technology has now progressed to the point where it is possible for a single-chip real-time graphics processor to support complex, user-programmable shading programs with high performance.

This transition in graphics hardware has enabled new applications and has changed existing ones. For visual simulation applications such as flight simulators, real-time programmable shading enables much greater visual realism. In entertainment applications such as games, programmable shading allows artists to create a unique "look." For data visualization applications such as volume rendering, programmable shading allows the transfer functions that map data values to colors to be more complex and allows these transfer functions to be modified interactively. Finally, programmable graphics hardware is sufficiently flexible that it can be used as a general-purpose parallel computer, to run applications that its designers didn't even anticipate.

Many of the sophisticated shading and lighting algorithms that were originally developed for offline rendering can now be used for real-time rendering. However, there are some significant differences between real-time programmable shading and offline programmable shading. A few of these differences are transitory and will disappear as graphics hardware continues to evolve. But many of them are more fundamental and are likely to persist for many years. One major goal of this chapter is to explain these fundamental differences and their implications—in other words, to provide a bridge between the established domain of offline programmable shading and the newer domain of real-time programmable shading. The other major goals of this chapter are to describe the current state of the art in real-time programmable shading and to provide examples of shaders that run on real-time hardware.

Team LRN

After this introduction, the remainder of this chapter is organized as follows:

- First we describe why graphics hardware is fast and the limitations that it imposes on programmability in order to provide this high performance.
- Then we provide an initial example of a real-time shading program and show that it can be written in two very different styles—one style is a RenderMan-like style, and the other style is closer to the underlying hardware representation of the program.
- The next two sections discuss the factorization of shading into "surface" and "light" computations and describe the interface between applications and shading programs.
- We then provide two longer shading programs that demonstrate useful real-time shading techniques.
- We conclude with some strategies for developing real-time shaders and some thoughts on the future of real-time programmable shading.

The remainder of this introductory section will cover three topics. First, we will summarize the fundamental differences between real-time shading and offline shading. Then, we will explain our reasons for using a high-level language to program real-time graphics hardware, instead of an assembly-level language. Finally, we will list some of the topics that are *not* covered in this chapter because they are very similar to the corresponding topics from offline procedural shading and are thus well covered earlier in this book and elsewhere.

What Makes Real-Time Shading Different?

Although real-time programmable shading is similar in many respects to offline programmable shading, there are several unique characteristics of real-time programmable shading:

- *Most applications are interactive.* As a result, the shader writer usually does not know which viewpoints will be used to view an object and may not even know which lights will be near an object.
- *Performance is critical.* Real-time rendering requires a greater emphasis on performance than offline rendering. Furthermore, performance must be consistent from frame to frame.

Team LRN

- *Shaders execute on graphics hardware.* Graphics hardware provides high performance at low cost, but imposes certain restrictions on shading programs in order to obtain this high performance.

We will discuss each of these points in more detail.

In offline rendering, the viewpoints used with any particular shader are known prior to the one-time final rendering, so shaders and lighting can be tuned for these viewpoints. In contrast, most interactive applications (e.g., games) give the user control over the viewpoint, either directly or indirectly. Thus, the shader writer does not know in advance what viewpoints will be used with the shader. As a result, the tasks of scene lighting and shader antialiasing are more difficult than they are in the offline case. Programmers must insure that their solutions work for any viewpoint, rather than just for a particular set of viewpoints that were specified by the movie director.

One likely consequence of this difference is that the adoption rate of physically based illumination techniques will be more rapid in real-time rendering than it has been in offline movie rendering.

Although real-time shading presents new challenges to the shader programmer, it also provides one very important advantage: Scenes can be re-rendered in fractions of a second rather than minutes or hours. As a result, it is much easier to experiment with changes to a shader until the desired visual effect is achieved. The importance of this rapid modify/compile/render cycle cannot be overstated. More than most other types of programming, shader development is inherently iterative because the algorithms being implemented are ill-defined approximations to complex real-world phenomena. The ultimate test for a shader is "Does it look right?" so the development process is most efficient when it can be rapidly driven by this test.

Rendering performance is an issue in almost any type of rendering, but it is especially important for real-time rendering. The final frames for a real-time application may be rendered billions of times. In contrast, the final frames for a movie that is generated with an offline renderer are only generated once. The economics of these two cases are very different, requiring that much more effort be expended on optimizing rendering performance for the real-time case.

The consequences of this difference are apparent when we examine the shader-writing styles for offline rendering and real-time rendering. Many offline shaders are designed in part as modeling tools. As a result, these shaders have many parameters and option flags and can be thousands of lines long. These shaders provide great flexibility to artists, but at a cost in complexity. In contrast, real-time shaders are

Team LRN

more highly specialized. Real-time shaders also rely more heavily on performance-tuning techniques such as the use of table lookups for vector normalization.

Real-time rendering imposes one additional performance requirement that is not present for offline rendering: the rendering cost of every frame must be below a certain threshold to provide acceptable interactivity. In contrast, in a computer-generated movie, it is acceptable for a few frames to take an order of magnitude longer to render than the average frame.

Offline shading systems use only the CPU, but real-time shading systems perform most of their computations on the graphics hardware (GPU), because the GPU's peak performance is about two orders of magnitude greater than that of the CPU. But this high performance comes with a penalty—the GPU programming model is more restricted than that of the CPU. These restrictions are necessary to allow programmable GPUs to provide high performance at a low cost. The section "Real-Time Graphics Hardware" describes these restrictions and explains how they enable high performance.

Why Use a High-Level Programming Language?

Most graphics hardware supports one or more low-level programming interfaces, generally at the assembly language level. It is possible to program the GPU using these low-level interfaces, but we won't describe them in this chapter. Instead, this chapter uses a high-level shading language for GPU programming. Programs written in this high-level language (often referred to as *shaders*) must be compiled into assembly language before being run.

A high-level shading language provides a number of advantages over an assembly language interface:

1. It is easier and more productive to program in a high-level language. Ease of programming is especially important when developing shaders because the best approach to shader development is to try something, look at it, and then iterate. In contrast, programming at the assembly language level is sufficiently painful that it discourages exploration of ideas.
2. Writing programs in a high-level language makes it easy to create "libraries" of shaders. More importantly, the shaders from such a library can be easily modified and/or combined to meet specific needs.
3. A high-level language provides at least some degree of hardware independence. In contrast, GPU assembly languages are often completely different for

Team LRN

products from different hardware companies, and in many cases are even different for different products from the same company.

4. A high-level language (and associated compiler) can virtualize the hardware to hide hardware resource limits. For example, a compiler can use multiple rendering passes to hide limits on the number of textures, to hide limits on the number of temporary variables, or to hide limits on the size of a shader. With the proper hardware support for this virtualization, it can be completely invisible to the programmer.

5. A high-level language has the potential to provide *better* performance than typical handwritten assembly language code. The reason is that the compiler can optimize shaders using detailed information about the GPU that would be too tedious to use when writing assembly language code by hand. When the compiler is developed by the hardware vendor, the compiler may also use information about the GPU design that the hardware vendor would be unwilling to disclose publicly for competitive reasons.

For these reasons, we expect that high-level languages will become the standard GPU programming interface in the future, and we focus on them in this chapter.

What You Need to Learn Elsewhere

We've already discussed some of the differences between real-time and offline programmable shading, but the two are also similar in many respects. Many of the features of real-time shading languages have been adopted from the RenderMan shading language, and many of the techniques used in offline shaders are equally useful in real-time shaders. Since there is already a wide variety of high-quality information available about offline procedural shading, this chapter largely avoids repeating this information. Instead, we encourage you to use these other resources, then rely on this chapter to learn about the unique characteristics of real-time shading.

For readers who are unfamiliar with procedural shading, Chapter 2 of this book provides essential background material. The important topics in Chapter 2 include the basic introduction to procedural shading; the distinction between "image-based" and "procedural" approaches to shading; and the discussion of antialiasing for procedural shaders.

The standard procedural shading language for offline rendering is the RenderMan shading language. Although current real-time shading languages differ in important ways from RenderMan, almost all of the basic shader-writing strategies

Team LRN

that are used for RenderMan shaders are equally applicable to real-time shaders. There are two books that describe the RenderMan shading language and discuss shader-writing techniques. The first is *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics,* by Steve Upstill (1990). The second is *Advanced RenderMan: Creating CGI for Motion Pictures,* by Tony Apodaca and Larry Gritz (2000). The *Advanced RenderMan* book is a particularly good complement to the material in this chapter.

There are several different implementations of the RenderMan standard. The first one is Pixar's PhotoRealistic RenderMan (PRMan). Another is the Blue Moon Rendering Tools (BMRT). Most RenderMan shaders that are simple will run on any implementation of RenderMan, and these shaders are a valuable source of ideas for real-time shading.

If you are writing a complete real-time graphics application, you will need to understand the entire graphics pipeline, not just the programmable shading parts of it. There are two major interfaces for controlling the entire graphics pipeline—Direct3D and OpenGL. Historically, OpenGL has been the better-architected API, but it often requires the use of vendor-specific extensions to access the latest hardware features. In contrast, Direct3D has been more difficult to learn and use, but has evolved more rapidly to support leading-edge features with a common interface. OpenGL is described in the *OpenGL Programming Guide* (OpenGL ARB 1999), and in extension specifications available from the Web sites of graphics hardware companies. A variety of books are published every year or two to describe the latest version of Direct3D.

Real-Time Graphics Hardware

High-performance real-time programmable shading languages are designed to run on GPUs. To make effective use of these languages and to be able to predict how their capabilities will evolve with time, it is crucial to have some understanding of graphics hardware. This section discusses some of the characteristics of graphics hardware that are most important for programmable shading.

CPUs and GPUs are designed with very different goals. CPUs are designed to provide high performance on general-purpose, *sequential* programs. In contrast, GPUs are designed to provide high performance for the *specialized* and *highly parallelizable* task of polygon rendering and shading.

GPU architectures are organized around these two themes of specialization and parallelization. For example, rasterization and texture decompression are typically performed using specialized hardware. As a result, the programmability of these

Team LRN

parts of the graphics pipeline is limited. For other parts of the graphics pipeline that are programmable, the GPU architecture imposes certain programming restrictions to facilitate massive parallelism at low cost. For example, the programmable pixel/fragment processors in 2002-generation GPUs do not support a store-to-memory instruction because supporting this type of memory access with reasonable ordering semantics would impose extra costs in a parallel architecture.

Object Space Shading versus Screen Space Shading

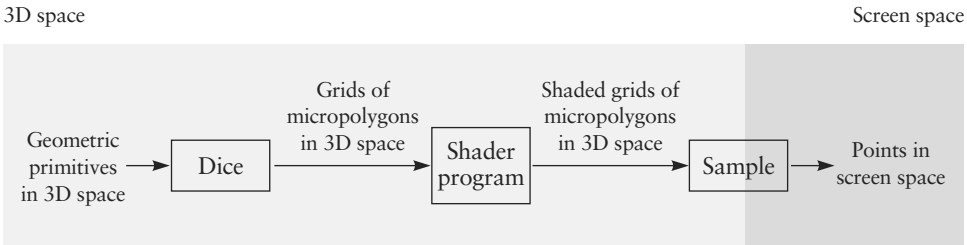
Hardware graphics pipelines perform some programmable shading in object space (at vertices) and some programmable shading in screen space (in effect, at pixels). In contrast, the REYES algorithm used by Pixar's PRMan performs all shading in object space, at the vertices of automatically generated *micropolygons*.[1] Hardware pipelines use the hybrid vertex/pixel shading approach for a variety of reasons, including the need for high performance and the evolutionary path that graphics hardware has followed in the past. We will explain the two approaches to programmable shading in more detail and discuss their advantages and disadvantages.

RenderMan uses curved surfaces (both tensor-product splines and subdivision surfaces) as its primary geometric primitives. To render these surfaces, PRMan dices them into grids, then uniformly tessellates each grid into "micropolygons" that are smaller than a pixel, as in Figure 3.1(a). The programmable shader executes at each vertex of these micropolygons. The programmable shader may change the position of the vertex, as well as calculate its color and opacity. Next, for each pixel in which a micropolygon is visible, its color is evaluated at one or more screen space sample points. The color at each sample point is determined using either flat shading or Gouraud shading. The algorithm is described in more detail in Cook, Carpenter, and Catmull (1987) and Apodaca and Gritz (2000).

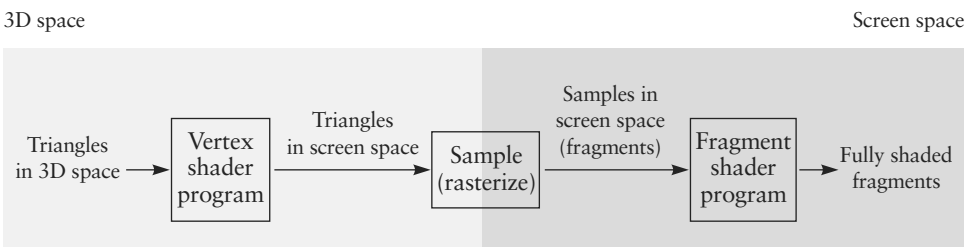
In contrast, graphics hardware uses triangular polygons as its primary geometric primitive. One set of programmable shading computations is performed at the vertices of these triangles, prior to their transformation into screen space. Then, the triangles are rasterized into screen space samples, referred to as *fragments*. Next, a second set of programmable computations is performed at each fragment. These two sets of programmable computations are commonly known as per-vertex and per-fragment computations, respectively. This process is illustrated in Figure 3.1(b). Finally, if multisample antialiasing is active, the visibility of a fragment may be tested at multiple points in screen space (not shown in the figure).

1. Note that other RenderMan implementations, such as BMRT, use different approaches.

Team LRN



(a)



(b)

FIGURE 3.1 Comparison of (a) the REYES pipeline (used in Pixar's RenderMan) and (b) the pipeline used in real-time graphics hardware.

The most important difference between object space shading and screen space shading is that an object space shader may change the position of the surface—that is, it can displace the surface, using either a displacement map or a procedural computation. In contrast, a screen space shader is forbidden to change the three-dimensional position of a fragment because the (u,v) screen space location of the fragment has already been fixed by the rasterizer. A screen space shader is limited to changing the screen space depth of a fragment.

For rapidly varying shading computations such as specular lighting, it is important that the shading computation be performed approximately once per pixel to produce high-quality images. Screen space shading is automatically performed at this rate, but object space shading is not. The REYES algorithm solves this problem by automatically generating micropolygons of the necessary size from curved surfaces specified by the user. There are a number of reasons why this approach has not yet been used in real-time hardware:

- Historically, the performance of graphics hardware has not been high enough to allow the use of pixel-sized polygons. Transformation and lighting were

Team LRN

performed relatively infrequently at the vertices of large polygons, and these results were interpolated in screen space. This interpolation is much more efficient than reevaluating the shading equation at each pixel. The performance of graphics hardware has now increased to the point where this optimization is less important than it once was, but because performance is so crucial in real-time graphics, it is still useful to be able to perform some computations less often than others.

- If all shading computations are performed at object space vertices, it is crucial that polygons be approximately the size of a pixel. Adopting this approach for real-time use would require automatic tessellation of curved surfaces and/or large polygons to form micropolygons. To avoid performance loss from the CPU-to-GPU bandwidth bottleneck, this tessellation must be performed by the graphics hardware. As of this writing, some hardware does support hardware tessellation of curved surfaces, but this hardware lacks the automatic adjustment of tessellation rate needed to guarantee pixel-sized micropolygons and has not been widely used by developers. Hardware implementation of adaptive tessellation is challenging; feedback paths of the type used in the REYES dicing algorithm are complex to implement, as are algorithms that avoid cracks at boundaries where tessellation rates change. However, we can expect that these challenges will eventually be overcome.
- Because graphics hardware has historically performed high-frequency shading computations at fragments rather than vertices, 2002-generation graphics hardware does not support the use of texture maps at vertices. A switch to 100% object space shading would require that texture mapping capability be added to the programmable vertex hardware.

There are several other differences between the pure object space shading model used by the REYES algorithm and the hybrid model used by graphics hardware:

- If a displacement map contains large, rapidly varying displacements, the object space shading model can create polygons that are large and flat-shaded in screen space. In contrast, the hybrid vertex/fragment shading model performs fragment computations once per pixel even for large polygons generated by the displacement process.
- The object space shading model allows motion blur and depth-of-field effects to be generated inexpensively. Shading computations are performed once in object space, and the shaded surfaces are sampled many times to generate the motion

Team LRN

blur and depth-of-field effects. In contrast, the accumulation buffer technique that is typically used to generate these effects in conjunction with screen space shading requires that the shading computations be repeated for each set of samples.

- Antialiasing of procedural shaders often relies on derivative computations, and computing these derivatives is very simple at fragments. If vertices are specified directly (rather than automatically generated by dicing, as in REYES), computing derivatives at vertices is more complex than doing so at fragments because the neighborhood relationships for vertices are less regular.

One minor, but sometimes annoying, implication of the different rendering approaches used by RenderMan and real-time hardware is that real-time hardware does not make the "geometric normal" available to vertex programs. In RenderMan, the geometric normal is available as the variable `Ng` and is directly computed from the local orientation of the curved surface being rendered. In 2002-generation graphics hardware, only the shading normal `N` is available in vertex programs. However, the geometric normal can be made available to fragment programs on some graphics hardware via two-sided lighting tricks.

Parallelism

Graphics hardware relies heavily on several forms of parallelism to achieve high performance. Typically, programming any type of parallel computer is difficult because most algorithms have dependencies among the different computations required by the algorithm. These dependencies require communication and synchronization between processors. The need for communication and synchronization tends to reduce processor utilization and requires expensive hardware support.

The programmable processors in graphics hardware largely avoid these costs by restricting the communication that is allowed. As a result, the hardware can provide very high performance at low cost, as long as the user's algorithms can tolerate the restrictions on communication.

On 2002-generation graphics hardware, the programmable vertex processor allows only access to data for a *single* vertex. As a result, the hardware can perform operations on different vertices in any order or in parallel. If multiple vertex processors operate in parallel, no communication or synchronization is required between them.[2] Likewise, the programmable fragment processor allows access to data for just

2. Note that some synchronization is required when vertices are assembled into primitives, but because this operation is not programmable, it can be performed by highly specialized hardware.

Team LRN

one fragment. High-level real-time shading languages that compile to this hardware must expose the restriction that each vertex is processed independently and each fragment is processed independently.

If both read and write access is provided to a memory, that memory provides another form of communication. The system must define (and support) ordering semantics for the reads and writes. To avoid the costs associated with supporting these ordering semantics, 2002-generation GPUs tightly restrict memory accesses. Fragment processors can read from texture memory, but not write to it. Fragment processors can write to frame buffer memory, but not read from it. The only read access to frame buffer memory is via the nonprogrammable read/modify/write blend unit. These read/modify/write operations are performed in the order that polygons were specified, but the hardware required to implement this ordering is simplified because the fragment processor is not allowed to specify the address of the write. Instead, the write address is determined by the (nonprogrammable) rasterizer. All of these restrictions are exposed in high-level real-time shading languages. Fortunately, the restrictions are straightforward and are also present in the RenderMan shading language.

One implication of the restricted memory access model is that a compiler must store all of a shader's temporary variables in hardware registers. On a CPU, temporary variables can be "spilled" from registers to main memory, but this simple strategy does not work on a GPU because there is no general read/write access to memory. The only way to perform this spilling on 2002-generation graphics hardware is for the compiler to split the shading computation into multiple hardware rendering passes. Then, temporary variables can be saved into the graphics memory in one pass and restored from that memory in a subsequent rendering pass. Note that the CPU software must issue a barrier operation between rendering passes to switch the region of memory used for the temporary storage from write-only access ("frame buffer mode") to read-only access ("texture mode").

At the fragment level, most 2002-generation graphics hardware uses a SIMD (single instruction, multiple data) computation model—the same sequence of instructions is executed for each fragment. At the instruction set level, the SIMD model is evident from the absence of a conditional branch instruction. Hardware that uses this SIMD computation model cannot execute data-dependent loops using a single hardware rendering pass. Shading languages that compile to this hardware must either forbid these operations or implement them inefficiently by using multiple rendering passes and stencil-buffer tests.[3] If/then/else operations can be executed within

3. This strategy is inefficient because it consumes additional memory bandwidth and requires retransformation and rerasterization of geometry.

Team LRN

a single rendering pass, but only by effectively executing both the "then" and "else" clauses for every fragment, and then choosing which result to keep.

The situation is different for vertex computations. Some 2002-generation hardware supports looping and branching for vertex computations. Thus, the vertex hardware uses an SPMD (single program, multiple data) computation model—different vertices execute the same program, but they may take different execution paths through that program.

The SIMD computation model is less expensive to implement in hardware than the SPMD model because the instruction storage, fetch, and decode hardware can be shared among many processing units. Furthermore, SIMD execution can easily guarantee that the computations for different fragments complete in the correct order. However, data-dependent looping is valuable for algorithms such as anisotropic filtering, antialiasing of turbulent noise functions, and traversal of complex data structures. For these reasons, it is likely that graphics hardware will eventually incorporate direct support for conditional branch instructions in fragment programs.

Hardware Data Types

Graphics hardware has historically used low-precision, fixed-point data types for fragment computations. The registers and computation units for smaller data types are less expensive than those for larger data types; for example, the die area required by a hardware multiplier grows approximately as the *square* of the number of bits of precision. But general-purpose shader programs require higher-precision data types, especially for texture coordinate computations (e.g., computing the index into an environment map). Shader programs are also easier to write if the hardware supports floating-point data types instead of fixed-point data types, since the availability of floating-point data types frees the programmer from having to worry about scale factors.

For these reasons, 2002-generation hardware supports 32-bit floating-point fragment arithmetic. However, because lower-precision arithmetic can be performed more quickly, 2002-generation hardware also supports lower-precision floating-point and fixed-point data types. Shaders that use these lower-precision data types will typically run faster than shaders that reply primarily on the 32-bit floating-point data type.

Pre-2002-generation graphics hardware only supports 8-bit or 9-bit fixed-point arithmetic for most fragment computations. If a shader that is written in a high-level language needs to run on this older hardware, its variables must all be declared using these older fixed-point data types. Even on newer hardware, shader writers must

Team LRN

restrict themselves to using data types that are hardware supported on all of the hardware platforms that they are targeting. Compilers can effectively hide some differences between hardware instruction sets, but they cannot efficiently hide differences between hardware data types.

Resource Limits

In general, graphics hardware has limits on resources of various types. These limited resources include the following:

- Memory for storing instructions
- Registers for storing temporary variables
- Interpolators for vertex-to-fragment interpolants
- Texture units
- Memory for textures and frame buffer

Ideally, these resource limits would be hidden from the high-level language programmer by some combination of hardware-based and software-based virtualization techniques. As an analogy, CPUs hide limits on main memory size by using a virtual memory system that pages data to and from disk. Performance may degrade when this virtualization is activated, but programs produce exactly the same results that they would if more main memory were available. CPU hardware includes features that are designed to support this virtualization. Unfortunately, 2002-generation graphics hardware does not yet support resource virtualization as well as CPUs. Most resource limits can be circumvented by using multiple rendering passes, but multipass rendering is an imperfect virtualization technique. Generally, the need to resend geometry from the CPU to the GPU cannot be completely hidden from the application program. Furthermore, multipass rendering produces incorrect results for overlapping, partially transparent surfaces. As a result, shader programmers must sometimes concern themselves with limits on the resources that a program may consume.

Even if virtualization is implemented well, it may cause nonlinear changes in performance that are sufficiently large to be of concern to programmers. For example, adding one more statement to a shader can cause its performance to drop in half on some graphics hardware. Programmers who are concerned about performance on particular hardware must understand the resource limits and performance characteristics of that hardware.

Team LRN

Memory Bandwidth and Performance Tuning

Z-buffered rendering consumes an enormous amount of memory bandwidth for reading and writing the frame buffer. If surfaces are texture mapped, additional memory bandwidth is required to read the texels from texture memory. As a result, the performance of real-time graphics hardware has historically been limited primarily by memory bandwidth, especially when rendering large polygons.

As VLSI technology advances, this limitation becomes more pronounced because on-chip computational performance improves more rapidly than bandwidth to off-chip memories. One solution to this problem would be to put the entire frame buffer, plus a frame-sized texture cache, on chip. But as of this writing, this solution is not yet economical for PC graphics hardware.

Graphics hardware designers have instead taken a different approach—-they have placed the increased computational performance at the programmer's disposal, by adding the programmable hardware units that are the focus of this chapter. Besides enabling the generation of much more realistic images, these programmable units indirectly reduce the demand for memory bandwidth by allowing programmers to use a single rendering pass to implement algorithms that used to require multiple rendering passes. In processor design terminology, GPUs have become a specialized type of stream processor (Rixner et al. 1998). Stream processors are designed to support a high compute-to-bandwidth ratio, by reading a packet of data (e.g., a vertex), executing a complex computational kernel (e.g., a vertex program) on this data, and then writing a packet of data (e.g., a transformed vertex).

Unfortunately, the availability of these programmable units makes performance tuning very complex. At any one time, rendering performance may be limited by vertex-processor performance, fragment-processor performance, memory bandwidth (including texture fetches), or any one of a whole set of other factors, such as host-to-GPU bandwidth. Detailed approaches to performance tuning are hardware dependent, but we will describe two performance-tuning techniques that are broadly applicable.

First, it is often possible to trade compute performance for memory bandwidth and vice versa. The simplest example of such a trade-off is the use of a table lookup (i.e., texture read) in place of a complex computation. If a shading program is compute limited, then making this trade-off will improve rendering performance.

Second, for programs that are limited by memory bandwidth, it may be possible to improve performance by more effectively utilizing the hardware's texture cache. In general, reducing the size of a texture will improve performance, as will restructuring programs that use table lookups so that they are more likely to repeatedly access the same table entries.

Team LRN

SIMPLE EXAMPLES

Throughout the rest of this chapter, we will use a series of example shading programs that show some of the effects that can be achieved with a real-time programmable shading system. These shaders are written using the Stanford real-time shading language. The Stanford real-time shading system is an experimental system designed to show that a hardware-independent shading language can be efficiently compiled to programmable graphics hardware. At the time that we developed the examples in this chapter, it was the only high level language available for mainstream programmable graphics hardware.

As this book was in the final stages of publication, commercially supported programming languages for programmable GPUs were beginning to appear. These languages, including NVIDIA's Cg language, follow the philosophy of the C language in the sense that they are designed primarily to provide convenient access to all of the capabilities of the underlying hardware, rather than to facilitate any particular use of the hardware. In contrast, RenderMan, and to a lesser extent the Stanford shading language, are designed for the specific task of surface shading and include a variety of features designed to support that task.

All but one of the examples in this chapter avoid most of these specialized features and therefore can be easily ported to commercially supported GPU programming languages such as Cg. However, you may want to check the author's Web site for updated versions of the examples before porting them yourself.

Vertex and Fragment Code in the Stanford Shading System

Most of the commercially available GPU programming languages require that the user write two separate programs—a vertex program and a fragment program. One of the unique features of the Stanford shading language is the ability to combine vertex and fragment computations within a single program. The user specifies whether computations are per vertex or per fragment by using the `vertex` and `fragment` type modifiers. The following code shows how a simple program can use these type modifiers. The Stanford language also allows computations to be performed once for each group of primitives, using the `primitive group` type modifier. Typically, this capability is used for operations such as inverting and transposing the model-view matrix. On 2002-era hardware, these operations are executed on the CPU rather than the GPU.

```
//
// A very simple program written in the Stanford shading language. This program
// includes both vertex computations and fragment computations. The fragment
```

```
// type modifier indicates that a variable is instanced once for each fragment.
// Likewise, the vertex and primitive group type modifiers indicate that a vari-
// able is instanced once for each vertex or once for each group of primitives,
// respectively.
//
// Scale 'u' component of texture coordinate, and apply texture to surface
//
surface shader float4
applytexture (vertex float4 uv, primitive group texref star) {
    vertex float4 uv_vert = {uv[0]*2, uv[1], 0, 1}; // Scale texcoord
    fragment float4 uv_frag = uv_vert;              // Interpolate
    fragment float4 surf = texture(star, uv_frag);  // Texture lookup
    return surf;
}
```

The Stanford language uses simple "type promotion" rules to determine whether specific computations within an expression are mapped to the CPU, the vertex processor, or the fragment processor. For example, when a `vertex` expression is multiplied by another `vertex` expression, the computation is performed by the vertex processor and yields another `vertex` expression. When a `vertex` expression is multiplied by a `fragment` expression, the `vertex` expression is interpolated to produce a `fragment` expression before performing the multiplication on the fragment processor. The compiler is responsible for using these rules to split the user's single program into separate vertex and fragment programs that can be executed on the graphics hardware, and a separate primitive group program that is executed on the CPU.

This unified vertex/fragment programming model is very convenient for straight-line code, but it becomes unwieldly in a language that supports imperative looping constructs, such as "for" and "while" loops. The Stanford language doesn't support these constructs, but the newer commercially available languages do, and therefore they require the user to write separate vertex and fragment programs. The example shaders in this chapter are designed to be easily split into separate vertex and fragment programs for such languages.

Two Versions of the Heidrich/Banks Anisotropic Shader

One of the most important advantages of programmable graphics hardware is that it can be used to implement almost any lighting model. For anisotropic surfaces, one lighting model that is especially appropriate for real-time use is Heidrich and Seidel's (1999) formulation of the Banks (1994) anisotropic lighting model. Figure 3.2 illustrates this lighting model applied to a sphere.

Heidrich and Seidel's lighting model was designed to execute efficiently on graphics hardware. The vertex-processing hardware computes a pair of dot

Team LRN

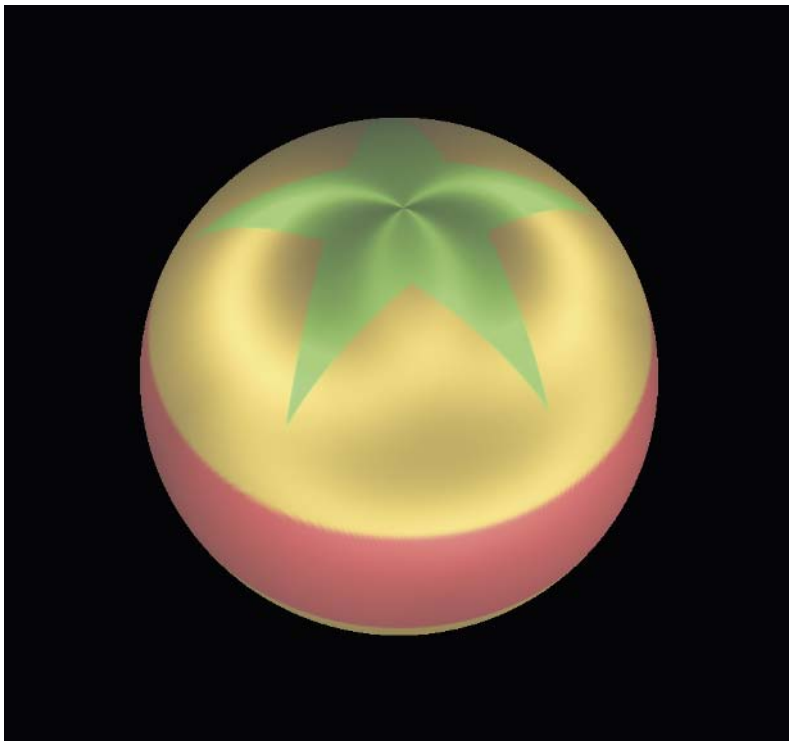


FIGURE 3.2 Banks/Heidrich anisotropic lighting shader applied to a sphere.

products, and these dot products are used as indices for a 2D table lookup at each fragment. The 2D table is precomputed and stored in a 2D texture map. Heidrich and Seidel have described a whole family of real-time lighting models that are based on the idea of factoring lighting models into independent terms that depend on only one or two angles. Each such term can be stored in a 1D or 2D texture map. The SIGGRAPH paper that describes their approach (Heidrich and Seidel 1999) is worthwhile reading for anyone who is interested in lighting models that are both realistic and efficient.

We will now look at two programs that implement the Banks/Heidrich lighting model using two different shader-writing styles. The first program expresses almost all of the transform and shading computations explicitly. For example, the first two statements transform the position from homogeneous model space to Cartesian eye space, and later statements transform the surface normal and binormal vectors (`Ndir` and `Bdir`) to eye space. These computations rely directly on parameters

Team LRN

provided through the geometry API, such as the vertex position and the modelview matrix.

```
//
// The first of two implementations of the Heidrich/Banks anisotropic lighting
// model. This implementation explicitly expresses every computation that will
// be performed by the programmable vertex and fragment hardware, except for the
// final transformation of position into clip space. The anisotex texture must
// be precomputed as described in Heidrich and Seidel (1999). Adapted from a
// shader written by Kekoa Proudfoot; used with permission.
//
// Relies on the following predefined variables:
//  __position      = object space position for vertex (from API vertex calls)
//  __normal        = object space normal for vertex   (from API vertex calls)
//  __binormal      = object space binormal for vertex (from API vertex calls)
//  __modelview     = 4 × 4 modelview matrix
//  __invmodelview3 = inverse transpose of 3 × 3 modelview matrix
//  __ambient       = global ambient color
//
surface shader float4
anisotropic1 (texref star,         // Surface texture
              texref anisotex,     // Precomputed table for BRDF calc
              float4 lightpos,     // Light position
              float4 lightcolor) { // Light color
    //
    // Determine light's location and its intensity at surface
    //
    vertex float4 Peye4 = __modelview * __position; // calc eye space obj pos
    vertex float3 Peye3 = {Peye4[0], Peye4[1], Peye4[2]} / Peye4[3];
    vertex float3 Lvec  = rgb(lightpos) - Peye3;
    vertex float3 Ldir  = normalize(Lvec);
    vertex float4 light_intensity = lightcolor / (1.0 + .01*length(Lvec));
    //
    // Look up surface texture using computed texture coordinates
    //
    vertex float4 surf_uv = {0.5*__position[2]+0.5, 0.5*__position[0]+0.5, 0, 1};
    fragment float4 surf  = texture(star, surf_uv);
    //
    // Compute eye space normal and binormal direction vectors
    //
    vertex float3 Ndir = normalize(__invmodelview3 * __objnormal);
    vertex float3 Bdir = normalize(__modelview3 * __objbinormal);
    //
    // Heidrich/Banks anisotropic lighting
    // Uses a texture as table lookup
    //
    vertex   float3 Edir     =normalize(-Peye3); // Direction from surface to eye
    vertex   float4 aniso_uv = {0.5*dot(Bdir,Edir)+0.5, 0.5*dot(Bdir,Ldir)+0.5, 0, 1};
```

Team LRN

```
    fragment float4 anisotrp = light_intensity *
                               max(dot(Ndir,Ldir),0) * texture(anisotex, aniso_uv);
    //
    // Calculate ambient term; modulate lighting by surface texture
    //
    float4 Ma = {0.1, 0.1, 0.1, 1.0}; // Ambient coefficient
    return surf * (Ma*__ambient + anisotrp);
}
```

This program also explicitly specifies whether variables are instanced at each vertex or at each fragment. This style of programming makes it clear which computations will be performed in the vertex-processing hardware, and which will be performed in the fragment-processing hardware. It also makes it easy to port this program to shading languages that require vertex and fragment computations to be specified in separate programs.

The second version of the Heidrich/Banks shader program looks quite different, although it compiles to the same hardware operations:

```
//
// The second of two implementations of the Heidrich/Banks anisotropic lighting
// model. This implementation uses separate surface and light programs and
// relies on implicitly specified computations. Two unique features of the
// Stanford system are used here: The perlight variables in the surface shader
// are instanced once for each active light shader. The integrate() function
// performs a sum over all lights of its argument. The argument to integrate()
// must be a perlight expression, but the result is an ordinary expression.
// Adapted from a shader written by Kekoa Proudfoot; used with permission.
//

//
// Light shader with constant and linear terms
//
// Relies on the following predefined variable:
//  Sdist = distance from light to surface point
//
light shader float4
simple_light (float4 lightcolor, float ac, float al)
{
    return lightcolor * (1.0 / (al * Sdist + ac));
}

//
// Heidrich/Banks anisotropic shader; works with any light shader(s).
//
// Relies on the following predefined variables:
//   Pobj  object space surface position, w=1
//   N     eye space normal vector, normalized, w=0
```

```
//   B     eye space binormal vector, normalized, w=0
//   E     eye space local eye vector, normalized, w=0
//   L     eye space light vector, normalized, w=0  (per light)
//   Cl    color of light                           (per light)
//   Ca    color of global ambient light
//
surface shader float4
anisotropic2 (texref star, texref anisotex) {
    //
    // Look up surface texture using computed texture coordinates
    //
    float4 surf_uv = {0.5*Pobj[2]+0.5, 0.5*Pobj[0]+0.5), 0, 1};
    float4 surf = texture(star, surf_uv);
    //
    // Heidrich/Banks anisotropic lighting
    // Uses a texture as table lookup
    //
    perlight float4 aniso_uv = {0.5*dot(B,E)+0.5, 0.5*dot(B,L)+0.5, 0, 1};
    perlight float4 anisotrp = Cl *
                               max(dot(N,L),0) * texture(anisotex, aniso_uv);
    //
    // Calculate ambient term; modulate lighting by surface texture
    //
    float4 Ma = {0.1, 0.1, 0.1, 1.0}; // Ambient coefficient
    return surf *(Ma * Ca + integrate(anisotrp));
}
```

The differences between the two versions of the program fall into three categories. First, the second version relies on the shading system to implicitly perform some computations, such as the transformation of the normal vector from object space to eye space. Second, this program specifies "surface" and "light" computations in two different shaders, and relies on the shading system to combine these computations into a single hardware program at compile time. We will discuss this capability of the Stanford system in more detail later. Third, this program doesn't explicitly specify whether computations are performed per vertex or per fragment. Instead, it relies on the Stanford system's default rules to determine whether computations are performed per vertex or per fragment. As mentioned earlier, these default rules are similar to the type promotion rules in the C language.

SURFACE AND LIGHT SHADERS

Historically, real-time graphics APIs have allowed the application to manage surface properties separately from light properties. Surface properties include the texturing modes and surface color(s). Light properties include the number of lights, the

type of each light, and the color(s) of each light. For example, in OpenGL, the `glColorMaterial` routine modifies surface properties, and the `glLight` routine modifies light properties.

This separability of surface and light properties corresponds to the behavior we observe in the physical world—any light in a scene will modify the appearance of any surface, as long as the two are visible from each other. The RenderMan shading language supports this distinction by providing two types of shaders—*light* shaders and *surface* shaders. A light shader is responsible for calculating the intensity and color of light that is incident on any surface element from a particular light. A surface shader is responsible for determining the final surface color (as seen from a particular direction), given a set of incident light intensities.

Unfortunately, light and surface computations are not cleanly separated in graphics hardware because a Z-buffered rendering pipeline is fundamentally just a surface renderer. If the application changes the number of active lights, the graphics system must change the computation performed by the hardware at every point on every surface. For a fixed-function rendering pipeline, the hardware driver can manage the required changes to the hardware configuration, since the fixed-function pipeline's lighting model is simple enough that the configuration changes are localized to the vertex-processing part of the pipeline.

When the hardware API supports full programmability, it becomes much more difficult for the hardware driver to maintain the illusion that surface and light computations are fully separable. As a result, 2002-generation low-level programmable APIs for graphics hardware eliminate the distinction between surface and light computations. These APIs expose one programmable vertex processor and one programmable fragment processor. These processors must perform all of the per-vertex and per-fragment computations, respectively. The user is responsible for combining surface and light computations to create these programs.

Even though the hardware does not directly support separate surface and light shaders, it is possible for high-level real-time shading language to do so. However, if a language supports separability, the separability is an illusion that must be maintained by the language's compiler and run-time system. The compiler and run-time system are responsible for combining each surface shader with the active light shader(s) to create a single vertex program and single fragment program that will run on the underlying hardware.

As with almost any binding operation, there is a trade-off between the performance of the combined code and the cost of the binding operation. If the binding is done early, prior to most of the compilation process, then the resulting code is very efficient, but the binding operation itself is expensive—it is essentially a recompile. If

Team LRN

the binding is done late, then the binding operation is inexpensive, but the resulting code is likely to be less efficient because the compiler cannot perform global optimizations.

The Stanford shading system supports separate surface and light shaders using the early-binding model. The binding process is explicit: the run-time API requires that light shaders be bound to a surface shader before compiling the surface shader. The Stanford system could have used an implicit binding model instead, but implicit early binding is dangerous because the binding process is sufficiently expensive that the application should be aware of it. In addition to supporting surface and light shaders, the Stanford system also supports deformation shaders, which can modify the position and local coordinate frame of a surface point.

As of 2002, most other high-level real-time shading languages do not support separate surface and light shaders. One reason for this omission is that built-in support for separate surfaces and lights requires that the shading system impose an interface between the two types of shaders. For example, in the RenderMan shading language, lights return a single RGB color. In the Stanford shading language, lights return a single RGBA color. Interestingly, neither of these interfaces is sufficient to support the OpenGL lighting model! The reason is that the OpenGL lighting model requires that lights return three RGBA colors—ambient, diffuse, and specular. Although such lights are not physically realizable, the flexibility they provide has proven to be very useful in practice.

Since it is difficult to pick a single surface/light interface that is appropriate for all uses, it is likely that future real-time shading languages will provide a more general mechanism for specifying interfaces between cooperating routines. In the RenderMan shading language, it is common for users to define such interfaces indirectly, using extra shader parameters and the RenderMan message-passing routines.

THE INTERFACE BETWEEN SHADERS AND APPLICATIONS

A shader program is not a stand-alone program. It operates on data (e.g., vertices) provided through some external mechanism, it obtains its parameters (e.g., light positions) through some external mechanism, and it is enabled and disabled through some external mechanism. Typically, this external mechanism is an API that is either layered on top of the primary real-time API or fully integrated with it. For example, the Stanford shading system uses a set of API calls that are implemented on top of OpenGL, while the proposed OpenGL 2.0 shading language requires changes to the OpenGL API itself. Although the RenderMan standard is best known for its shading

Team LRN

language, it also includes an entire API that is used to specify geometry, to set shader parameters, and to generally control the rendering process.

For programmers who migrate from fixed-function real-time graphics pipelines to programmable pipelines, it is useful to understand which parts of the fixed-function API are subsumed by shading programs and which are not. Most of the state management API routines are replaced by the shading language. For example, a shader program replaces the fixed-function API routines used to configure texture blending, to configure lighting, and to set texture coordinate generation modes. However, current shading languages do not replace the API commands used to load textures. The application program is responsible for loading textures before using a shader.

A programmable shading system retains the fixed-function API routines used to feed data down the graphics pipeline and augments them with new routines. For example, the API routines used to specify the position and normal vector of a vertex are retained. New API routines are provided to allow the specification of shader-specific vertex parameters (e.g., skinning weights) and to allow the specification of shader-specific state (e.g., "time," for an animated shader).

To use a more concrete example, the Stanford shading system provides API routines that allow an application to do the following:

- Load a shader's source code from a file
- Associate a light shader with a surface shader
- Compile a surface shader and its associated light shader(s)
- Bind to a compiled shader (i.e., prepare to render with it)
- Specify values of arbitrary shader parameters (either through immediate-mode style commands or through vertex arrays)
- Specify vertices to be rendered (again, either through immediate-mode style commands or through vertex arrays)

The following program shows the API calls required to render one face of a cube using the Stanford shading system.

```
//
// Pieces of a C program that make the API calls required to render one face of a
// cube using the Stanford shading system. All of the sgl* calls are shading API
// calls. sglParameterHandle() binds a numeric handle to a string name.
// sglParameter4fv() specifies the value of a shader parameter, using a numeric
```

Team LRN

```
// handle. sglBindShader() specifies which shader should be used for rendering.
// The other sgl* calls are wrappers around the similarly named OpenGL calls.
// Note that this listing omits the API calls required to initialize the shading
// system and to compile the shader.
//
// Setup
//
float green[] = {0.0, 1.0, 0.0, 1.0};
float a[]     = {0.0, 0.0, 0.0, 1.0};
float b[]     = {0.0, 1.0, 0.0, 1.0};
float c[]     = {1.0, 1.0, 0.0, 1.0};
float d[]     = {1.0, 0.0, 0.0, 1.0};
//
// Assign numeric handles to parameter names
// Only needs to be done once for a shader
// This shader requires a "surfcolor" parameter and a "uv" parameter.
//
const int UV        = 3;
const int SURFCOLOR = 4;
sglBindShader(200); // Bind a shader that was compiled earlier
sglParameterHandle("uv",        UV);
sglParameterHandle("surfcolor", SURFCOLOR);
.
.
.

//
// Render y=1 face of a cube
// Surface color is green; assign "uv" coordinates for texturing
//
sglBindShader(200); // Bind a shader that was compiled earlier
sglBegin(GL_QUADS);
sglParameter4fv(SURFCOLOR, green);
sglNormal3f(0.0,1.0,0.0);
sglParameter4fv(UV, a);    sglVertex3f(-1.0, 1.0, -1.0);
sglParameter4fv(UV, b);    sglVertex3f( 1.0, 1.0, -1.0);
sglParameter4fv(UV, c);    sglVertex3f( 1.0, 1.0,  1.0);
sglParameter4fv(UV, d);    sglVertex3f(-1.0, 1.0,  1.0);
sglEnd();
```

It is important to remember that most programmable shaders require significant support from the main application program. In addition to binding the shader, the application program must specify values for the shader's parameters, load textures, and provide any auxiliary data that is needed in conjunction with each vertex and/or triangle. When designing a shader, it is crucial to decide how much of a burden you are willing to impose on the application program. For example, will you restrict

Team LRN

your shader to using a specific set of values that is already provided with each vertex by the application? Or are you willing to require that the application provide new per-vertex values just for your shader?

MORE EXAMPLES

This section discusses two shading programs that are substantially more complex than the previous examples. The first is a shader used for volume rendering. It requires hardware capabilities such as dependent texturing that were not supported on PC graphics hardware prior to NVIDIA's GeForce3. The second is a shader for procedurally generated flame. It consists entirely of fragment operations and requires an NVIDIA NV30 (fall 2002) or better to run.

Volume-Rendering Shader

Now that consumer-level graphics cards support both 3D textures and fragment-level programmability, it is possible to use these cards to perform volume rendering with a variety of classification and lighting functions. This type of volume rendering is implemented by rendering a series of 2D slices from a 3D texture and compositing the slices as they are rendered (Cabral, Cam, and Foran 1994; Van Gelder and Kim 1996). The volume shader is applied to each slice as it is rendered and composited.

The following program is an example of this type of shader. Figure 3.3 illustrates the use of this shader. Many variants on this theme are possible; for example, better-looking results can be obtained by using a double-sided lighting model in place of the single-sided lighting model used in the `lightmodel_gradient()` function.

```
//
// A volume-rendering shader written in the Stanford shading language.
// On each invocation, the shader classifies and shades one sample from the
// volume. Information about the volume is stored in 3D texture maps. This
// shader was written by Ren Ng; used with permission.
//
// functions to transform 3 and 4 vectors from
// eye space to object space
//
surface float3 objectspace(float3 V) {return (invert(affine(__modelview))*V);}
surface float4 objectspace(float4 V) {return (invert(__modelview)*V);}

// simple light shader
//
light shader float4
```

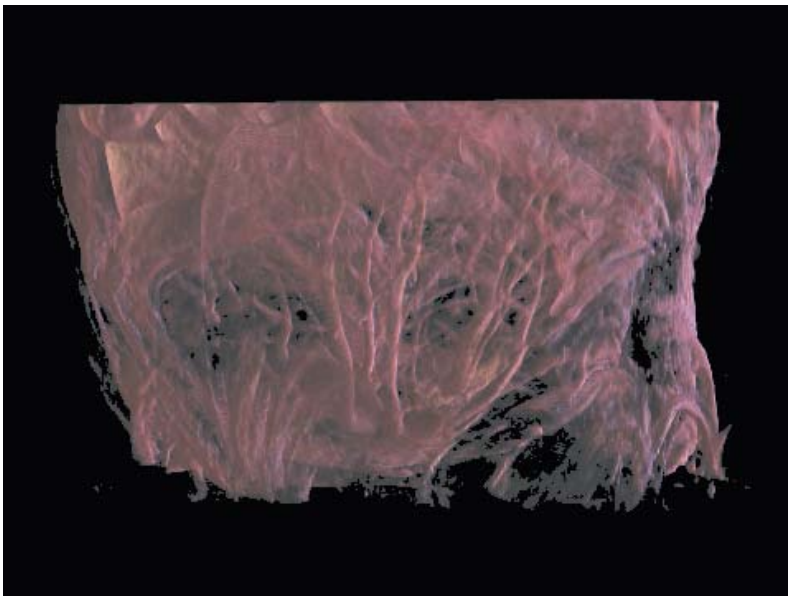


FIGURE 3.3 Using a volume-rendering shader to visualize a 3D MRI of a mouse abdomen. Ren Ng wrote the shader and captured this frame. The mouse data set is courtesy of G. A. Johnson, G. P. Cofer, S. L. Gewalt, and L. W. Hedlund at the Duke University Center for In Vivo Microscopy.

```
constant_light(float4 color) {
  return color;
}

// function to do specular & diffuse lighting based on a gradient field
//
// Hobj: the half angle vector in object space
// Lobj: the lighting vector in object space
// Nobj: the gradient vector in object space
// a: ambient term
// d: diffuse coefficient
// s: specular coefficient
//
surface float3
lightmodel_gradient (perlight float3 Hobj,
                     perlight float3 Lobj,
                     fragment float3 Nobj,
                     float3 a, float3 d, float3 s) {
    // Diffuse
    perlight float NdotL = dot(Nobj, Lobj);
    perlight float3 diff = d * clamp01(NdotL);
```

```
    // Specular exponent of 8.0
    perlight float NdotH  = clamp01(dot(Nobj, Hobj));
    perlight float NdotH2 = NdotH  * NdotH;
    perlight float NdotH4 = NdotH2 * NdotH2;
    perlight float NdotH8 = NdotH4 * NdotH4;
    perlight float3 spec  = s * NdotH8;

    // Combine
    return integrate(a + rgb(Cl) * (diff + spec));
}

// Surface shader for resampling polygons drawn through a volume.
// Note that compositing is set up by the calling application.
//
// density_plus_gradientmag: a 2-component, 3D texture containing
//    density in the red channel and the magnitude of gradient in alpha
// gradient: a 3-component, signed 3D texture containing the gradient
// color_opacity_transfer2d: a 2D texture that contains an RGBA value
//    (a base color and opacity), to be indexed by gradient magnitude
//     and density
// voxelsPerSlice: inverse of the average number of resampling slices
//    that pass through each voxel
// ambientColor: ambient lighting color
// specularColor: specular lighting color
// objToTex: a matrix that transforms from the object coordinates of
//    the volume to texture coordinates. This matrix provides the client
//    application with flexibility in exactly how the volume is
//    stored in an OpenGL texture
// opacityFactor: a dial for the application to easily change the
//    overall opacity of the volume.
//
surface shader float4
volume_shader (texref density_plus_gradientmag,
               texref gradient,
               texref color_opacity_transfer2d,
               primitive group float voxelsPerSlice,
               primitive group float4 ambientColor,
               primitive group float4 specularColor,
               primitive group matrix4 objToTex,
               primitive group float opacityFactor) {
  // texture coordinate set-up: convert from world
  // coordinates to object coordinates.
  matrix4 worldToTex = objToTex * invert(__modelview);
  float4 uvw         = worldToTex * P;

  // Classification: map (density, gradient magnitude) -> (base color, alpha)
  //                     using a dependent texture lookup
  fragment float4 density_maggrad = texture3d(density_plus_gradientmag, uvw);
  fragment float4 dep_uv = { density_maggrad[3], density_maggrad[0], 0, 1 };
  fragment float4 basecolor_alpha = texture(color_opacity_transfer2d, dep_uv);
```

Team LRN

```
  fragment float3 basecolor = rgb(basecolor_alpha);
  fragment float alpha = basecolor_alpha[3] * (4.0 * voxelsPerSlice);

  // Shading: Blinn-Phong model, evaluated in object space
  //  3D texture gives an object space normal vector
  //  transform eye space H and L vectors into object space
  fragment float3 Nobj = rgb(texture3d(gradient, uvw)); // uses signed texture
  perlight float3 Hobj = normalize(objectspace(H));
  perlight float3 Lobj = normalize(objectspace(L));
  float3 A = rgb(ambientColor);
  float3 D = basecolor;
  float3 S = rgb(specularColor);
  fragment float3 color = lightmodel_gradient(Hobj, Lobj, Nobj, A, D, S);

  float opacFact = clamp01(opacityFactor);

  fragment float4 rgba = {opacFact * (4.0 * alpha) * color, // RGB
                          opacFact * (4.0 * alpha) }; // ALPHA
  return rgba;
}
```

Noise-Based Procedural Flame

The following program is an animated flame shader. This shader is sufficiently complex that it is at the outer extreme of what is reasonable in a real-time shader in 2002. Figure 3.4 shows a single frame generated using this shader.

```
//
// An animated flame shader consisting entirely of per-fragment computations.
// This shader is written in the Stanford shading language and is designed to
// be applied to a single square polygon with (u,v) in the range [0,1]. It
// compiles to 122 NV_fragment_program instructions. This shader is ©2001
// NVIDIA; used with permission.
//
float abs(float x) {
  return(select(x < 0, -x, x));
}

//
// fastspline--Evaluate spline function
//             for spline parameters (1, 0.8, 0.1, 0, 0)
//
float fastspline(float x) {

  float t0 = x*2;
  float r0 = 0.8 + t0*(-0.45 + t0*(-0.8 + t0*(0.55)));
```

FIGURE 3.4 One frame generated using an animated flame shader. This scene consists of only three rectangles—one for the stone wall, one for the floor, and one for the flame. The rectangle with the flame shader is rendered last because the flame is partially transparent and because the flame shader uses the depth of the background at each pixel.

```
  float t1 = (x-0.5)*2;
  float r1 = 0.1 + t1*(-0.4 + t1*(0.55 + t1*(-0.25)));

  float r = select(x < 0.5, r0, r1);
  return r;
}

// Rotate 2D vector (stored in float3) by 30 degrees and scale by 2
float3 rotate30scale2(float3 x) {
```

```
  return {x[0]*2.0*0.866, x[1]*2.0*(-0.5), 0.0} +
         {x[0]*2.0*0.5,   x[1]*2.0*(-0.866), 0.0};
}

// Return one 3D noise value from a 2D noise texture
float noise3D(float3 T, texref noisetex,
              float3 L1offset, float L1weight,
              float3 L2offset, float L2weight) {
  float L1 = texture(noisetex, T+L1offset)[0];
  float L2 = texture(noisetex, T+L2offset)[0];
  float N = L2*L2weight + L1*L1weight; // Range of N is [0,1]
  return abs(N-0.5);
}

// Obtain background depth for this pixel from a texture
// that has been created with render-to-texture or equivalent.
// In this case, the depth is packed into the RGB values.
float lookup_olddepth(texref depthtex) {
    float4 dtex = texture(depthtex, rgb(xyz_screen())/1024);
    return dtex[0] + dtex[1]/256 + dtex[2]/(256*256);
}

//
// Notes on texture parameters:
//   * 'noisetex' holds random noise.
//     Its wrap mode must be set to REPEAT for both dimensions.
//   * 'permutetex' holds a 1D array of random values
//     Its wrap mode must be set to REPEAT, and it can be point-sampled.
//   * 'depthtex' holds the depth of the background,
//     with the floating-point depth value stuffed into RGB
//
surface shader float4
flame (float4 uv, primitive group float time,
       texref permutetex, texref noisetex,
       texref depthtex) {
  fragment float u = uv[0];
  fragment float v = uv[1];
  //
  // Calculate the coordinates (T) for turbulence computation
  //
  float framediff = time*0.2; // Animate--upward flame movement with time
  float3 T = {u, v+framediff, 0};
  //
  // Scale the turbulence coordinates.
  // 'freqscale' adjusts the spatial frequency of the turbulence
  // The 1/64.0 scales into the range of the 64 × 64 noise texture
  //
  constant float freqscale = 16;
  T = T * ({freqscale, freqscale/2, 0} * {1/64.0, 1/64.0, 0});
```

Team LRN

```
//
// To get a third (temporal) dimension of noise, we generate a pair of
// pseudorandom time-varying offsets into the 2D noise texture. We use
// a 64 × 1 (really 64 × 64) permutation texture to generate the offsets.
//
constant float changerate = 6.0;
float timeval  = time*changerate;
float timerem  = timeval - floor(timeval);
float timebase = floor(timeval); // Determines pair of samples
float L1weight = 1.0 - timerem;
float L2weight = timerem;
float ocoord1  = timebase/64.0 + 1.0/128.0;
float ocoord2  = ocoord1 + 1.0/64.0;
float3 L1offset = rgb(texture(permutetex, {ocoord1, 1.0/128.0, 0.0}));
float3 L2offset = rgb(texture(permutetex, {ocoord2, 1.0/128.0, 0.0}));


//
// Generate turbulence with four octaves of noise
//
float turb;
turb = noise3D(T, noisetex, L1offset, L1weight, L2offset, L2weight);
T    = rotate30scale2(T); // Rotate T by 30 deg and scale by 2
turb = turb +
       0.5 * noise3D(T, noisetex, L1offset, L1weight, L2offset, L2weight);
T    = rotate30scale2(T); // Rotate T by 30 deg and scale by 2
turb = turb +
       0.25 * noise3D(T, noisetex, L1offset, L1weight, L2offset, L2weight);
T    = rotate30scale2(T); // Rotate T by 30 deg and scale by 2
turb = turb +
       0.125 * noise3D(T, noisetex, L1offset, L1weight, L2offset, L2weight);
//
// Calculate the flame intensity in the "edge" (silhouette) region.
// It decreases both with the distance from the vertical axis
// and with height. It is also perturbed by the turbulence value.
//
float turbscale = 0.5 + 0.7*(1-v);
float x = (1.0/sqrt(v)) * (abs(2.0*u-1.0) + turbscale*turb);
float edgedensity = 12.5 * fastspline(clamp(x, 0, 1));
//
// Calculate the color for the edge region of the flame
//
float FlameTemperature = 0.6;
float3 FlameColor = {1.0, FlameTemperature, FlameTemperature-0.2};
float3 edgecolor  = FlameColor * edgedensity;
//
// Calculate the color for the interior region of the flame
// The flame interior is cooler near the top
//
float indensity = (2.85*turb+0.55) + v*0.35;
```

Team LRN

```
  float3 incolor  = FlameColor * indensity;
  //
  // Transition from the interior color to the edge color at the point
  // where the densities (and thus colors) are equal
  //
  float3 flamecolor = select(edgedensity > indensity, incolor, edgecolor);
  float density     = select(edgedensity > indensity, indensity, edgedensity);
  //
  // Clamp the color and density
  //
  flamecolor = clamp(flamecolor, 0, 1);
  density    = clamp(density, 0, 1);
  //
  // If no depth-based attenuation is desired, can exit the shader here:
  // return {flamecolor, density};
  //
  // ** Depth-based attenuation **
  // This attenuation reduces the "straight edge" effect where the flame
  // meets the floor. It is equivalent to a very simple volume
  // integration (in Z)
  //
  constant float depthscale     = 0.002;
  constant float flamethickness = depthscale; // flame thick in [0,1] Z units
  constant float depthperturb   = 1.75 * depthscale; // depth perturb scale
  constant float edgeperturb    = 3.5 * depthscale; // depth perturb at side
  float olddepth = lookup_olddepth(depthtex);
  //
  // Perturb current depth by turbulence value (to avoid uniformity);
  // and by abs(u-0.5) to give rounded appearance to flame base
  //
  float depth = xyz_screen()[2];
  depth = depth + depthperturb*turb + edgeperturb*abs(u-0.5);
  //
  // Attenuation is proportional to difference between
  // current depth (after perturbation) and background depth.
  //
  float atten = (olddepth - depth) / flamethickness;
  return {flamecolor, density*min(atten,1.0)};
} // flame
```

This shader is almost entirely procedural; although it uses three textures, these textures are used only to assist in the generation of pseudorandom noise and to provide the shader with the depth of the background geometry at the current pixel.

Pseudorandom 3D noise can be generated using a variety of techniques. At one extreme is an almost entirely procedural technique (used by most of the examples in earlier chapters in this book); at the other extreme is a single 3D texture lookup.

Team LRN

This flame shader uses an intermediate approach: for each 3D noise evaluation, the shader performs a pair of two 2D texture lookups and combines the results. In effect, one dimension of the 3D noise function is evaluated procedurally, and the other two dimensions are obtained from a texture. The best approach to evaluating a 3D pseudorandom noise function depends strongly on the performance characteristics of the target hardware, so this shader's approach will not be ideal for all hardware.

This flame shader consists of four basic parts. First, the shader computes a turbulence function from four octaves of pseudorandom noise. The pseudorandom noise varies as a function of `u`, `v`, and time. The dependence on time animates the flame. Second, the shader calculates the flame's intensity. The spatial dependence of this computation provides the flame with its inverted-V shape. The shader uses this intensity to determine the location of the flame's silhouette and to calculate the flame's color in the regions near the silhouette. Third, the shader calculates the flame's color in the interior (nonsilhouette) regions of the flame. This color is primarily based on the pseudorandom turbulence value, but is adjusted to make the flame look hotter at the base and cooler at the top. Finally, the shader reduces the opacity of the flame if the depth of the flame's polygon is similar to the depth of the background object. Without this depth-based attenuation, the flame would end abruptly where the flame's polygon intersects the floor.

STRATEGIES FOR DEVELOPING SHADERS

Developing complex shaders is difficult and is somewhat different from most other programming tasks. The task is partly a programming task and partly an artistic task. A real-time shader needs to achieve a visually pleasing result with the best possible performance, and it is often not clear what the best strategy is for reaching this goal. The *Advanced RenderMan* book (Apodaca and Gritz 2000) is worth reading for its insights into this process.

The following hints may also be useful:

- Find photographs and/or videos of the real-world effect you are trying to model. For example, I found several video clips and photographs of real fire before writing the preceding flame shader.
- Don't start from scratch. Find an existing shader that does something similar to what you are trying to do, and use that as a starting point. Almost all of the examples in this chapter used some other shader as a starting point. Offline shaders can often be adapted for real-time use, or at least mined for insights

Team LRN

into the phenomenon that is being modeled. For example, the preceding flame shader evolved from a much simpler offline shader that I found in the ShaderLab2 package sold by Primitive Itch software. Graphics hardware companies usually maintain Web pages that provide examples of real-time shaders that run well on their hardware.

- Iterate! When you are working on a real-time shader, you have the luxury of being able to tweak it and see the result within seconds. Take advantage of this capability to look at the images produced by your shader as you develop it. Iteration is especially important for shaders that use a primarily phenomenological strategy (i.e., the computation is designed to look right), rather than a physically based strategy (the computation is designed to model some underlying physical process).
- Consider different approaches: in some cases an image-based approach (texture map) is best; in other cases a procedural approach works best. Most shaders use a combination of the two techniques. The light/surface interaction shader that John Carmack uses in his Doom game is an excellent example of combining computation with texture mapping to efficiently produce a desired effect on 2002-era graphics hardware.
- If shader performance is a major concern, make sure you understand which operations are efficient on the target hardware and which are not. Sometimes it can be helpful to examine the assembly language output from the compiler (if this is possible) to get ideas for improving the performance of a shader.

FUTURE GPU HARDWARE AND PROGRAMMING LANGUAGES

In 1999, GPUs were totally unprogrammable. Less than three years later, in 2002, GPUs included very general programmability for both vertex and fragment operations. Future changes are likely to be somewhat less dramatic, but we will continue to see an expansion of the generality and scope of GPU programmability. As GPU programmability matures, we can expect that hardware vendors will converge toward a common set of hardware data types and basic programmable features. This convergence will make it easier to use a high-level programming language to write shaders that are portable across a broad range of graphics hardware.

As graphics hardware becomes more general, it is likely that GPU programming languages will continue to evolve to look more like general-purpose programming

Team LRN

languages and less like specialized shading languages. This distinction is subtle, but important. General-purpose C-like languages provide straightforward access to hardware capabilities, without making any assumptions about the kind of program that the user is writing. In contrast, shading languages make certain assumptions about the kind of code that the user is writing. For example, the RenderMan shading language includes data types for points, vectors, and normals; in some cases, the compiler automatically transforms variables from one coordinate system to another. These capabilities are convenient for writing a shader, but unnecessary and even awkward for writing other types of GPU code. Ultimately, it is likely that specialized languages like RenderMan will be available for GPUs, but that programs written in these languages will be automatically converted into a C-like language before being compiled and run.

As graphics hardware and programming languages become more general, we will also see a wider variety of algorithms implemented on GPUs. For example, ray-tracing algorithms can be implemented on 2002-generation GPUs with reasonable efficiency (Purcell et al. 2002). If nontraditional uses of the GPU such as ray tracing can be shown to be sufficiently valuable, they will in turn influence the future evolution of GPUs.

LITERATURE REVIEW

The RenderMan shading language (Hanrahan and Lawson 1990) is the standard for offline programmable shading. The definitive definition of the language is provided by the RenderMan specification (Pixar 2000). The two books on RenderMan mentioned earlier in the chapter (Upstill 1990; Apodaca and Gritz 2000) contain much information that is useful for real-time shading programmers.

Prior to the era of programmable PC graphics hardware, there were two major efforts to build real-time programmable shading systems: the PixelFlow project Olano and Lastra 1998; Leech 1998) and SGI's OpenGL Shader (Peercy et al. 2000). Many of the lessons learned from these systems are still valuable for anyone who is designing new programmable hardware or shading languages.

The first shading language for PC graphics hardware was developed as part of the Quake III game engine (Jaquays and Hook 1999). This simple language showed that game writers would be willing to use a shading language if it provided portability to different hardware platforms without an unreasonable performance penalty. The Stanford shading system was the first system designed to target highly programmable PC graphics hardware and is described in Proudfoot et al. (2001).

Team LRN

ACKNOWLEDGMENTS

Eric Chan, Matt Pharr, Kurt Akeley, and David Ebert read drafts of this chapter and made excellent suggestions for improvements.

The contents of this chapter were inspired by research done as part of the Stanford real-time programmable shading project. Kekoa Proudfoot, Pat Hanrahan, and I began building the Stanford real-time shading system in 1999. Svetoslav Tzvetkov, Eric Chan, Pradeep Sen, John Owens, and Philipp Schloter added key capabilities to it. Ren Ng demonstrated that the system could be used to enable interactive, programmable volume rendering, in collaboration with David Ebert, Marc Levoy, and other project members. Our industry partners ATI (Andy Grueber, Steve Morein, Evan Hart, and Jason Mitchell), NVIDIA (Matt Papakipos, Mark Kilgard, David Kirk, and many others), SGI (Mark Peercy and Marc Olano), SONY-Kihara Research Center (Takashi Totsuka and Yuji Yamaguchi), SUN (Michael Deering), and 3dfx (John Danskin, Roger Allen, and Gary Tarolli) provided information about future graphics hardware, access to prerelease hardware and drivers, and funding. Matt, Mark, and others at NVIDIA were particularly strong supporters of our project. Their comments had an influence on the research direction we chose, and they provided us with outstanding access to prerelease hardware and drivers.

More recently, Tim Purcell, Ian Buck, Pat Hanrahan, and I began investigating the use of programmable graphics hardware for ray tracing and scientific computation. My thoughts about future graphics hardware architectures have been influenced by Bill Dally's Imagine project (Rixner et al. 1998; Owens et al. 2000) and by discussions with Mark Horowitz and Kurt Akeley.

I'd like to thank Pat Hanrahan for inviting me to work in a great research environment at Stanford, for sharing his thoughts on procedural shading, and for letting me take the time to write this chapter. Finally, I thank NVIDIA, and David Kirk in particular, for allowing me to include up-to-date information about NV30 hardware in this chapter, and for granting permission to include the flame shader that I wrote for NVIDIA.

Team LRN

4

CELLULAR TEXTURING

STEVEN WORLEY

Procedural texturing uses fractal noise extensively. This book has multiple chapters that are virtually subtitled "More Applications of Fractal Noise." The major reason for this popularity is that noise is very *versatile*.

The noise function simply computes a single value for every location in space. We can then use that value in literally thousands of interesting ways, such as perturbing an existing pattern spatially, mapping directly to a density or color, taking its derivative for bump mapping, or adding multiple scales of noise to make a fractal version. While there are infinitely many functions that can be computed from an input location, noise's random (but controlled) behavior gives a much more interesting appearance than simple gradients or mixtures of sines.

This simple functional nature of noise makes it an adaptable tool that you might call a texture "basis" function. A basis function should be a scalar value, defined over $\Re^3$. A good basis function is useful in the same way noise is, since its value can be used in versatile, diverse methods like we do with noise. Certainly not all textures use basis functions; a brick wall pattern is not based on mapping a computed value into a color spline.

This brings us to the introduction of new basis functions based on *cellular texturing*. Their appearance and behavior are very different than noise's (which makes them a good complement to the noise appearance), but they are basis functions, like noise, and we can immediately use many of the fun ideas we've learned about for noise for the new basis.

Cellular texturing is related to randomly distributed discrete features spread through space. Noise has a "discoloration" or "mountain range" kind of feeling. Cellular textures evoke more of a "sponge," "lizard scales," "pebbles," or "flagstones" feeling. They often split space into small, randomly tiled regions called cells. Even though these regions are discrete, the cellular basis function itself is continuous and can be evaluated anywhere in space.

Team LRN

The behaviors of noise and cellular texturing complement each other. This chapter starts with a description of the definition of the cellular function to teach you enough to use it as a texture author. The second half of the chapter is about implementing the basis function itself. Since source code for implementation is included with this book, this section can be skipped if you wish. However, the cellular basis is a lot more "hackable" than noise's definition, so many advanced textures will rewrite parts of the algorithm; therefore, so a complete description of the classic implementation is provided.

THE NEW BASES

The cellular texturing basis functions are based on the fundamental idea of randomly scattering "feature points" throughout 3D space and building a scalar function based on the distribution of the points near the sample location. We'll define this main idea with a few simple functions.

For any location **x,** there is some feature point that lies closer to **x** than any other feature point. Define $F_1(\mathbf{x})$ as the distance from **x** to that closest feature point. Figure 4.1 shows an example of this in 2D. As **x** varies, F_1 varies continuously as the distance between the sample location and the fixed feature point varies. It's still continuous even when the calculation "switches" from one feature point to its neighbor that has now become the closest. The *derivative* of F_1 will change discontinuously at these boundaries when the two feature points are equidistant from the sample location.

These locations where the function F_1 "switches" from one feature point to the next (where its derivative is discontinuous) are along the equidistance planes that

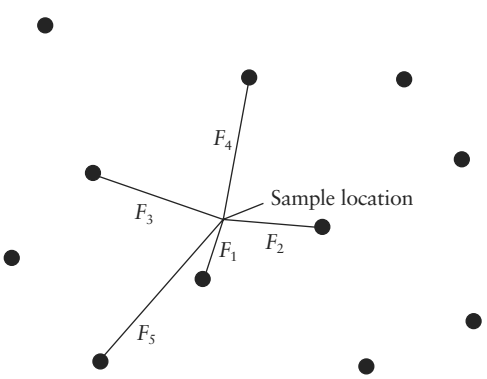


FIGURE 4.1 F_n values are the distance to the nth closest feature point.

Team LRN

separate two points in 3D space. These planes are exactly the planes that are computed by a Voronoi diagram, a partition of space into cellular regions.

The function $F_2(\mathbf{x})$ can be defined as the distance between the location $\mathbf{x}$ and the feature point that is the *second* closest to $\mathbf{x}$. With similar arguments as before, F_2 is continuous everywhere, but its derivative is not at those locations where the second-closest point swaps with either the first closest or third closest. Similarly, we can define $F_n(\mathbf{x})$ as the distance between $\mathbf{x}$ and the nth closest feature point.

The functions F have some interesting properties. F_n are always continuous. F_n are nondecreasing; $0 \leq F_1(\mathbf{x}) \leq F_2(\mathbf{x}) \leq F_3(\mathbf{x})$. In general, $F_n(\mathbf{x}) \leq F_{n+1}(\mathbf{x})$ by the definition of F_n. The gradient of F_n is simply the unit direction vector from the nth closest feature point to $\mathbf{x}$.

These careful definitions are very useful when we want to start making interesting textures. As with the noise function, mapping values of the function into a color and normal displacement can produce visually interesting and impressive effects. In the simplest case, $F_1(\mathbf{x})$ can be mapped into a color spline and bump. The character of F_1 is very simple, since the function increases radially around each feature point. Thus, mapping a color to small values of F_1 will cause a surface texture to place spots around each feature point—polka dots! *F*igure 4.2 shows this radial behavior in the upper left.

Much more interesting behavior begins when we examine F_2 and F_3 (upper right and lower left in Figure 4.2). These have more rapid changes and internal structure and are slightly more visually interesting. These too can be directly mapped into colors and bumps, but they can also produce even more interesting patterns by forming linear combinations with each other. For example, since $F_2 \geq F_1$ for all $\mathbf{x}$, the function $F_2(\mathbf{x}) - F_1(\mathbf{x})$ is well defined and very interesting, as shown in the bottom right of the figure. This combination has a value of 0 where $F_1 = F_2$, which occurs at the Voronoi boundaries. This allows an interesting way to make a latticework of connected ridges, forming a veinlike tracery.

We have interesting patterns in the basis functions F_1, F_2, F_3 and now we see that the linear combination $F_2 - F_1$ forms yet another basis. This leads us to experiment with other linear combinations, such as $2F_3 - F_2 - F_1$ or $F_1 + F_2$. In fact, most linear combinations tend to be interesting! The best way to see the different possible appearances is to generate multiple different linear combinations randomly and simply look at them, and save the ones that appeal to you. A user interface that lets you edit the linear coefficients is also fun, but a simple button to generate random coefficients is better. (It seems more useful to see brand-new patterns each time than to edit four mystery sliders that are difficult to "aim" to any goal.)

F_4 and other high n start looking similar, but the lower values of n (up to 4) are quite interesting and distinct. More importantly, linear combinations of these F_n

Team LRN

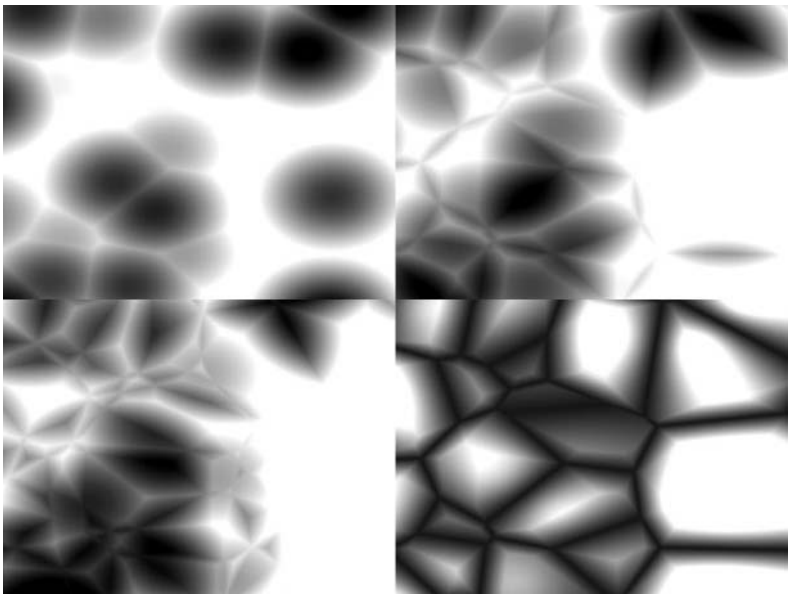


FIGURE 4.2 Gradient-mapped samples of F_1, F_2, F_3, and $F_2 - F_1$.

have more "character" than the plain F_n, particularly differences of two or more simple bases. Since the range of the function will change, it's easiest to evaluate 1000 samples or so and learn the minimum and maximum ranges to expect, and remap this value to a more stable 0–1 range. For "final" textures, this normalization only has to be precomputed once and hardwired into the source code afterwards.

Figure 4.3 shows 20 sample surfaces that are all just examples of combinations of these low-n basis functions ($C_1F_1 + C_2F_2 + C_3F_3 + C_4F_4$ for various values of C_n).

These patterns are interesting and useful, but we can also use the basis functions to make *fractal* versions, much like noise is used to produce fractal noise. By computing multiple "scales" of the function at different weights and scaling factors, a more visually complex appearance can be made. This is a simple loop, computing a function $G_n = \sum 2^{-i} F_n(2^i \mathbf{x})$ for moderate ranges of i ($i = 0$–5), and using G_n as the index for colors and bumps.

The fractal versions of any of the basic basis function combinations become extremely appealing. Figure 4.4 shows a fractal version of F_1 forming the spotted pattern and bumps on the hide of a creature. Fractal noise is used for the tongue, and a linear gradient is applied to the main body for color variation. Other fractal versions of primitives are shown in the row of cut tori in Figure 4.5.

Team LRN

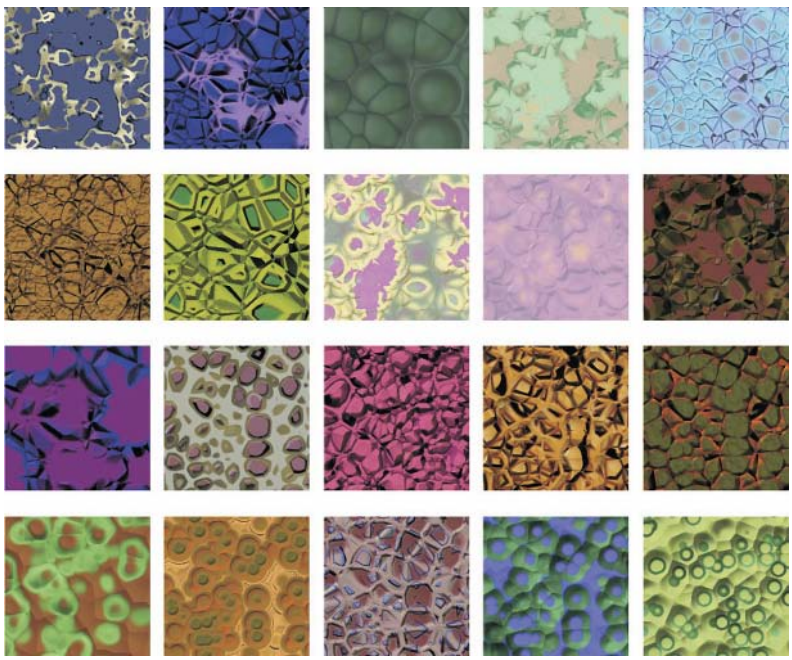


FIGURE 4.3 A variety of example appearances formed by linear combinations of the F_n functions.

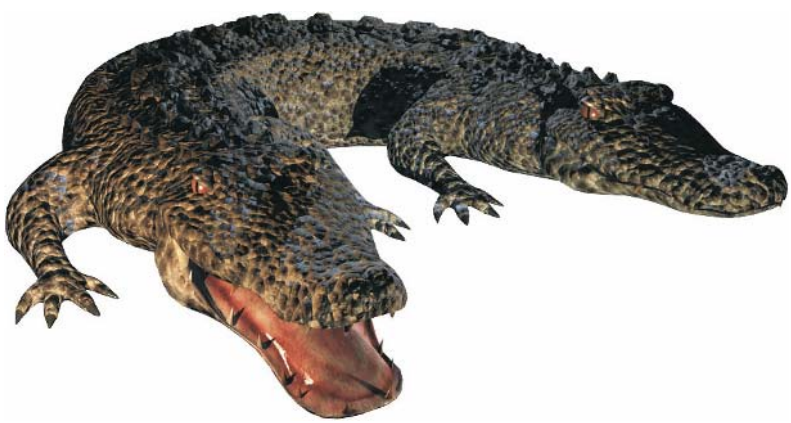


FIGURE 4.4 Natural-looking reptile hide using fractal-versions of the F_n functions.

Team LRN

FIGURE 4.5 More examples of fractal combinations.

The fractal version of F_1 is perhaps the most useful. Applied solely as a bump map, the surface becomes crumpled like paper or tinfoil. This surface has been extremely popular with artists as a way to break up a smooth surface, providing a subtle roughening with an appearance unlike fractal noise bumps. A surprising discovery was that a reflective, bumped-map plane with this "crumple" appearance bears an excellent resemblance to seawater, as shown in Figure 4.6. This bump-only fractal texture has become extremely popular in many renderers.

Since the cellular texture is a *family* of bases, it's fun to try more advanced incestuous combinations! Nonlinear combinations of simple polynomial products such as F_1F_2 or $F_3^2 - F_2^2$ are also interesting and useful texture bases. Again, renormalizing by empirically testing the output range makes the new basis easier to apply to color maps.

If the F_1 function returns a unique ID number to represent the closest feature point's identity, this number can be used to form features that are constant over a cell, for example, to shade the entire cell a single constant color. When combined with bumping based on $F_2 - F_1$, quite interesting flagstonelike surfaces can be easily generated. Figure 4.7 shows this technique, which also uses fractal noise discoloration in each cell. Note that unlike Miyata (1990), no precomputation is necessary and the surface can be applied on any shaped 3D object.

Bump mapping of the flagstonelike areas is particularly effective, and it is cheap to add since the gradient of F_n is just the radial unit vector pointing away from the appropriate feature point toward the sample location.

IMPLEMENTATION STRATEGY

It's not necessary to understand how to implement the cellular texture basis function in order to use it. But more than noise, the basis seems to encourage modifications and adaptations of the main algorithm to produce new effects, often with a very

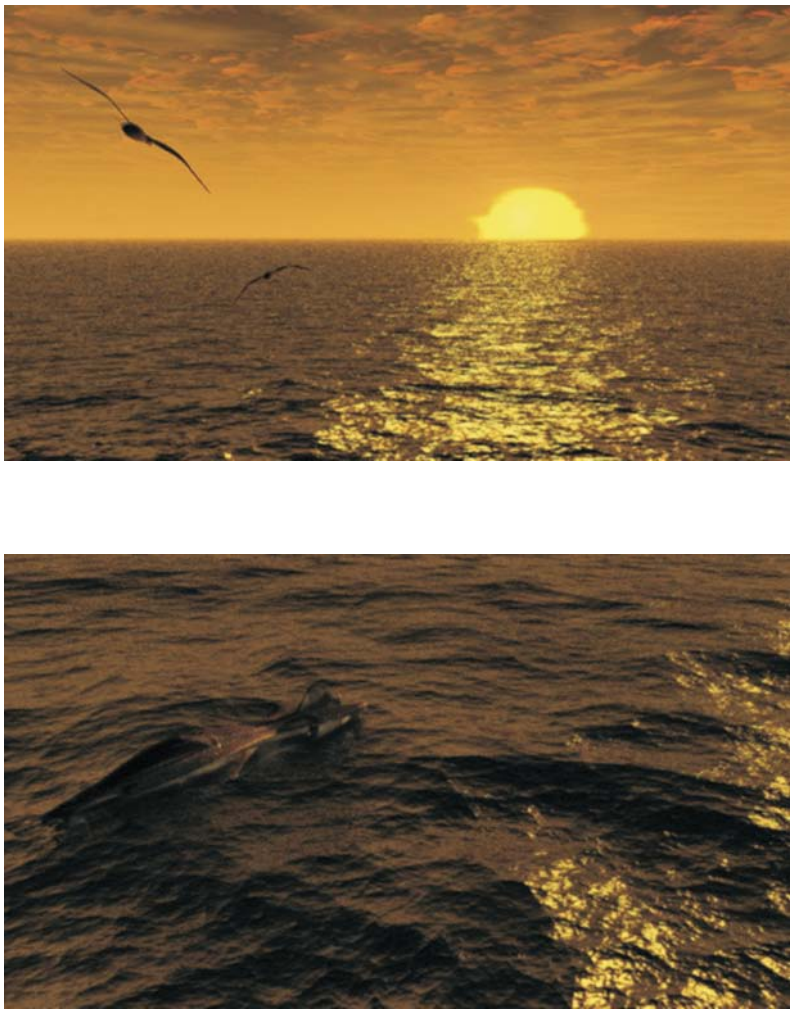


FIGURE 4.6 Sea surface formed from bump-mapped fractal F_1 functions.

different behavior than the original version. The following sections describe my implementation method, hopefully to allow you to modify it to make your own alternatives. The source code is also commented, but like all software, understanding the algorithm first will make understanding the actual code much easier.[1]

1. If you do create interesting extensions or speedups, please contact me at *steve@worley.com!*

Team LRN

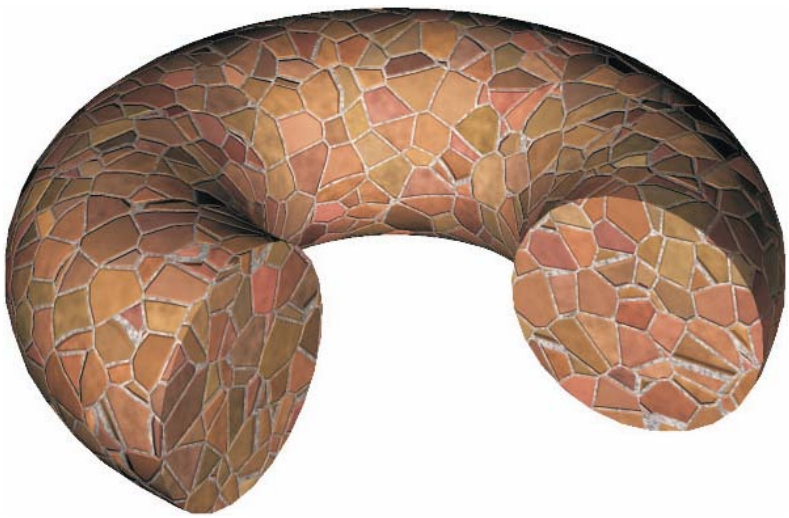


FIGURE 4.7 3D flagstone texture shows Voronoi cells.

The first step is to define how feature points are spread through space. The density and distribution of points will change the character of the basis functions. Our first assumption is that we want an *isotropic* function, to avoid any underlying pattern aligned with the world's axes. A simple idea like adding a point at every integer gridpoint and jittering their locations is easy to implement, but that underlying grid will bias the pattern, and it will be easy to see that "array" point layout.

The correct way to eliminate this bias is to keep the idea of splitting space into cubes, but choosing the number of points *inside* each cube in a way that will completely obscure the underlying cube pattern. We'll analyze this separately later.

Dicing Space

Since space will be filled with an infinite number of feature points, we need to be able to generate and test just a limited region of space at a time. The easiest way to do this is to dice space into cubes and deal with feature points inside each cube. This allows us to look at the points near our sample by examining the cube that the sample location is in plus the immediate neighbor cubes. An example is shown in Figure 4.8, where the "X" marks our sample location and dots show feature points in each cube. We can ignore the more distant cubes as long as we're assured that the feature points will lie within the 3×3 grid of local cubes.

Team LRN

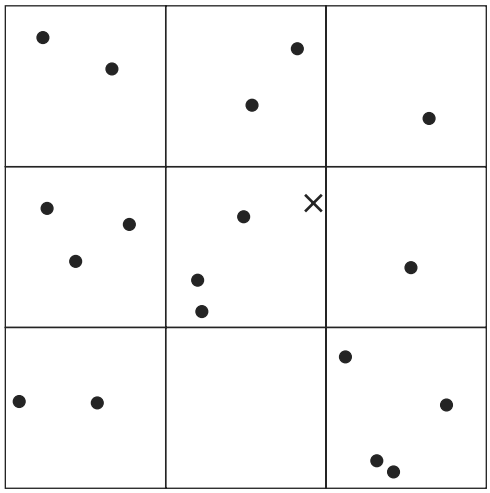


FIGURE 4.8 Searching for local feature points in neighboring cubes of space.

Each "cube" in space can be uniquely represented by its integer coordinates, and by simple floor operations we can determine, for example, that a point like (1.2, 3.33, 2.3) lies within the cube indexed by (1, 3, 2).

Now we determine how many and where feature points lie inside this cube. The "random" number we use to determine the number of points in a cube obviously must be unique to that cube and reproducible at need. There is a similar requirement in the noise function, which also uses a cubic lattice with fixed values associated with each gridpoint. We also need this seed to generate the location of the feature points themselves.

The solution to this problem is to hash the three integer coordinates of a cube into a single 32-bit integer that is used as the seed for a fast random number generator. This is easy to compute as something like $702395077x + 915488749y + 2120969693z \bmod 2^{32}$. The constants are random but chosen to be odd, and not simple multiples of each other mod 2^{32}. Like linear congruential random number generators, the *low-order* bits of this seed value are not very random.

We compute the number of points in the cube using this seed to pick a value from a short lookup table of 256 possibilities. This hardwired array of possible point populations is carefully precomputed (as described on page 145) to give us the "keep the points isotropic" magic property we desire. We use the *high-order* bits from our seed to index into the table to decide how many feature points to add into the cube. Since our table is a nice length of 256, we can just use the eight high-order

Team LRN

bits of the seed value to get the index. (The original 1996 paper used a different method for picking the point count, but the lookup table is both easier and more versatile.)

Neighbor Testing

Next, we compute the locations of the m feature points inside the sample cube. Again, these are values that are random, but fixed for each cube. We use the already initialized random number generator to compute the XYZ locations of each of the feature points. These coordinates are relative to the base of the cube and always range from 0 to 1 for each coordinate.

As we generate each point, we compute its distance to the original function evaluation location and keep a sorted list of the n smallest distances we've seen so far. As we test each point in turn, we effectively do an insertion sort to include the new point in the current list. This sounds expensive, but for typical values of n of 2 or 3, it only takes one or two comparisons and is not a major contributor to the algorithm's evaluation time.

This procedure finds the closest feature points and the values of F_1 to F_n for the points within the current cube of space (the one that contains the hit point). However, the feature points within a *neighboring* cube could quite possibly contain a feature point even closer than the ones we have found already, so we must iterate among the boundary cubes too. We could just test all 26 immediate neighboring cubes, but by checking the closest distance we've computed so far (our tentative nth closest feature distance) we can throw out whole rows of cubes at once by deciding when no point in the neighbor cube could possibly contribute to our list.

Figure 4.9 shows this elimination in a 2D example. The central cube has three feature points. We compute F_1 based on the feature point closest to the sample location (marked by "X"). We don't know yet what points are in the adjoining cubes marked by "?," but if we examine a circle of radius F_1, we can see that it's possible that the cubes labeled A, B, and C *might* contribute a closer feature point, so we have to check them. If we want to make sure our computation of F_2 is accurate, we have to look at a larger circle and additionally test cubes D and E. In practice, we can make these decisions quickly since it's easy to compare the current F distance to the distance of the neighbor cubes (especially since we can just compare the squared distances, which are faster to compute).

This kind of analysis also shows us that we need sufficient feature point density to make sure that one layer of neighbor cubes is enough to guarantee that we've found the closest point. We *could* start checking cubes that are even more distant

Team LRN

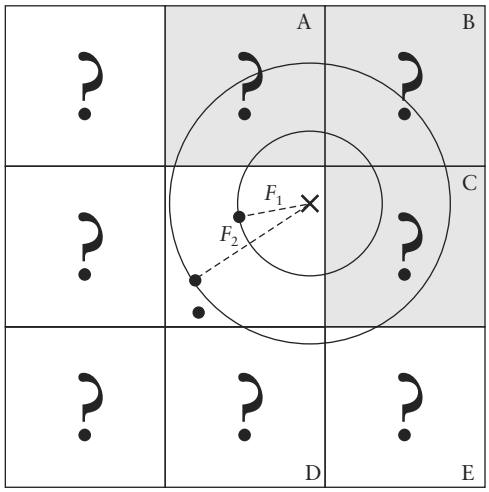


FIGURE 4.9 We don't need to test neighbor cubes that are too distant.

neighbors, but that becomes more and more expensive. Checking all neighbors requires at most $3^3=27$ cubes, but including more distant neighbors 2 cubes away would need $5^3=125$ cubes.

After we've checked all of the necessary neighbors, we've finished computing our values of *F*. If we computed F_n, we were effectively finding values for all F_1, F_2, . . . , F_n simultaneously, which is very convenient. The code included with this book returns all these values, plus the delta vectors corresponding to each feature point, plus a unique ID integer for each point (equal to the hashed cube ID plus the index of the feature point as it was computed). These all tend to be useful when using the function to form textures. If you don't need the extra information, you can modify the source code to return only the F_n values to gain a small speedup.

The Subtle Population Table

The desire for an isotropic distribution of points in space requires careful design. It can be done by choosing the number of points in each cube randomly but using a *Poisson distribution*. This distribution gives the probability of finding a specific number of points in a region when given a mean density. There may be more or less than this expected number of points in the region; the exact probabilities of any number of points in a region can be computed by using the discrete Poisson distribution function. If we generate the points inside of each cube randomly, with the

Team LRN

population based on the Poisson probabilities, the feature points will be truly isotropic and the texture function will have no grid bias at all.

Each cube in space may contain zero, one, or more feature points. We determine this on the fly quite simply by noting that the Poisson random distribution function describes the exact probabilities of each of the possible number of feature points occurring in the cube. For a mean density of λ feature points per unit volume, the probability of m points occurring in a unit cube is $P(\lambda,m) = \{(\lambda^{-m}\, e^{\lambda}\, m!)\}^{-1}$. Thus we can tabulate the probabilities for $m = 0, 1, 2, 3, \ldots$ easily. Computation is aided by the convenient recurrence relations $P(\lambda, 0) = e^{-\lambda}$ and $P(\lambda, m) = \frac{\lambda}{m} P(\lambda, m-1)$.

The value of λ to use is an interesting design decision. A high λ will tend to have a lot of feature points in each cube, and it will take extra time to generate and test the points. But a low λ will tend to have to evaluate more of the neighbor cubes. What's the best balance for speed? We're aided by the fact that we can change the feature scale of our function by simply scaling the input location by a constant. So no matter what λ we choose, we can renormalize the function to hide the specific λ. For the convenience of texture authors, I like to normalize the function such that the mean value of F_1 is equal to 1.0. This is similar to Perlin's decision to make the noise function vary over a characteristic distance of 1.0.

The obvious way to choose λ is to simply try different values and find which one gives the fastest average evaluation speed! But we do have an extra limitation. If λ is too low (less than 2.0 or so), then it's possible that the feature points are so sparse that the central 27 cubes are not enough to guarantee that we've found the closest point. We could start testing more distant cubes, but it's much more convenient to just use a high enough density to insure that the central 27 are sufficient. Also, notice that if we require the accurate computation of higher-order n values of F_n, we'll need a higher density of points to keep those more distant points within our "one cube" distance limit. Experiments (and some quick analysis) show that computation of an accurate F_n requires a density of $\lambda \approx n$. If the density is too low, our function fails and starts to sometimes return values for F_n that are discontinuous across the cube boundaries. This is not good.

Unfortunately, this lower limit for λ affects our efficiency, since by Murphy's law, fastest evaluation usually happens at lower λ values. So in practice, we can cheat by taking our carefully computed Poisson distribution and corrupting it to try to reduce the cases that cause problems. These are the cases when the population m is a low value of 0 or 1 (it's the low densities that don't give us enough candidates, and those empty cubes are particularly unhelpful). We could just *clip* the lowest allowable density to be at least 1. This helps evaluation efficiency enormously, since it allows us to use lower values of λ without causing the artifacts of discontinuous boundaries.

Team LRN

However, this clipping *violates our careful isotropic function design*. But if we choose to do this, we can at least do it knowledgeably and know that we can restore isotropy if we want to spend a little more CPU time to do it.

For actual implementation, we generate a set of integers that follow this Poisson density. By randomly selecting from this precomputed array, we can quickly compute cube populations that follow our chosen distribution. Since this table is precomputed, we can tweak and tune it as much as we like.

In practice, I've compromised speed versus isotropy by first generating a distribution following an ideal Poisson distribution. I then *bias* the distribution against those evil low populations by randomly increasing a fraction of the 0 and 1 populations by one. I can run lots of empirical tests to look for discontinuities indicating the density isn't sufficient and also at what average speed the function runs. For accurate computation of F_2, using a density of $\lambda = 1.60$ and incrementing 75% of the 0 and 1 populations is enough to prevent visible artifacts and give a good evaluation speed. Since most texturing usually only uses F_1 to F_4, we can tune for F_4 and find that $\lambda = 2.50$ and again a 75% random increment works. This is the precomputed table that's used in the example code for this book. There's also commented code for generating these lookup tables.

This population distribution seems overanalyzed, but the end results are worth it. While it's possible to simply use a constant population of, say, $m = 3$ and skip the table completely, the subtle axis bias can show up as noticeable artifacts that are difficult to analyze and understand. The Poisson table takes negligible extra evaluation overhead but gives you a noticeably higher-quality basis.

Extensions and Alternatives

The cellular noise function itself is extremely extendable and customizable. This is in contrast to Perlin noise, which tends to be viewed as a black box; you rarely have to open the hood to tinker with its engine. Even multifractals, introduced by Ken Musgrave, still use noise's basic function in its basic form.

A variety of small speedups can be made to the basic cellular basis algorithm and implementation, including making a version hardwired to return only specific F_n values, using fixed-point computation, using parallel vector CPU instructions such as MMX and SSE, changing the boundary cube segmentation to use hexagonal packing, and more.

Most fundamentally, the most interesting modification is to change the *definition of distance*. There are many different distance metrics that can be used in mathematics, and the Euclidean distance is just the simplest. The basis can instead be

Team LRN

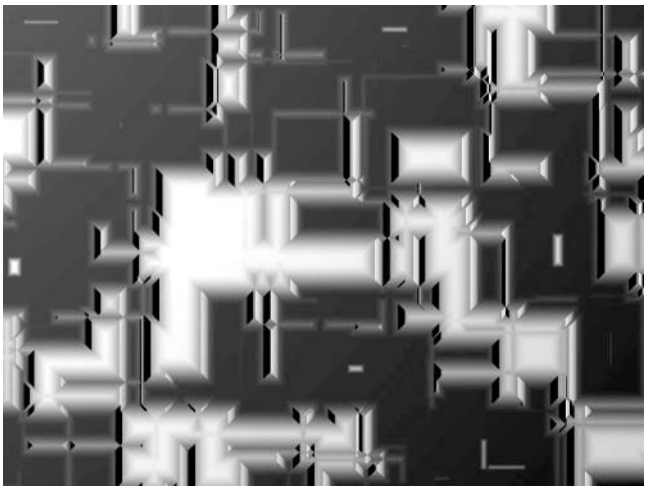


FIGURE 4.10 The F_1 "Manhattan" distance metric shows random rectangular shapes.

computed using the "Manhattan distance" between two points ($|dx| + |dy| + |dz|$), which forms regions that are rigidly rectangular but still random. These make surfaces like random right-angle channels—useful for spaceship hulls or circuit board traces. Figure 4.10 shows a nonfractal version of F_1 that uses this Manhattan distance metric. A radial coordinate version can cover spheres, creating a surface similar to the "Death Star." A whole range of cell shapes can be computed with a distance formula of the form $C_x dx^{n_x} + C_y dy^{n_y} + C_z dz^{n_z}$. Even more metrics are possible, including strange ones such as adding a small fractal noise variation to the Euclidean distance to make an oddly undulating craterscape. With these alternative metrics, the basis values are still monotonic ($F_{n+1} \geq F_n$), but the derivatives can change based on the metric you choose, which may affect your bump-mapping method.[2]

The main design constraint is that the distance metric should tend to increase as the Euclidean distance increases. This limitation is just to allow our simple "search all 27 neighboring cubes" algorithm to find the feature points within its search region. Since different metrics have different "reaches" that are not always easy to analyze, we usually use a simpler version of the basic algorithm that uses a larger

2. In practice, you can use a finite-difference bump-mapping method to allow arbitrary bumping without having to worry much about whether the derivative is easy to compute or not. See the discussion on page 170.

Team LRN

average density λ and checks all 27 neighbor cubes exhaustively. This means computation is slower than the optimized Euclidean distance version. After you've designed a new basis, you may be able to decrease λ and/or implement a fast rejection technique tailored to your new pattern. It's useful to make a test harness to evaluate your function millions of times randomly to measure speed as well as look for discontinuities in high F_n, which indicate you'll need a higher density for the limited 27-cube search.

Changes to other parts of the basic algorithm can produce other kinds of effects. The density of the feature points can be made to vary spatially, allowing for small, dense features in one area and larger features in another. Object geometry might be used to disperse precomputed feature spots (similar to Turk 1991) at the expense of requiring precomputation and table lookup, but gaining object surface dependence similar to the advantages Turk found. The algorithm is normally computed in 3D, but 2D variants are even faster; 4D variants can be used for animated fields, although these become very slow because of the necessity to compute up to $3^4=81$ cubes instead of $3^3=27$.

SAMPLE CODE

The Web site for this book (*www.mkp.com/tm3*) contains my implementation of the cellular texturing function, as well as a utility for generating the Poisson lookup tables. It is written in vanilla C for maximum portability. It is designed to return accurate values of F_n up to and including $n = 4$. It can return higher-order values as well, but in that case you may want to replace the Poisson table with a higher density to keep it artifact free. The function returns the values of F_n, the separation vector between the sample location and the nth feature point, and a unique ID number for that feature point. You are welcomed and encouraged to use this code in your own projects, private and commercial.

```
/* Copyright 1994, 2002 by Steven Worley
   This software may be modified and redistributed without restriction
   provided this comment header remains intact in the source code.
   This code is provided with no warrantee, express or implied, for
   any purpose.

   A detailed description and application examples can be found in the
   1996 SIGGRAPH paper "A Cellular Texture Basis Function" and
   especially in the 2003 book Texturing & Modeling, A Procedural
   Approach, 3rd edition. There is also extra information on the Web
   site http://www.worley.com/cellular.html.
```

Team LRN

```
   If you do find interesting uses for this tool, and especially if
   you enhance it, please drop me an email at steve@worley.com.

   An implementation of the key cellular texturing basis
   function. This function is hardwired to return an average F_1 value
   of 1.0. It returns the <n> closest feature point distances
   F_1, F_2, .. F_n the vector delta to those points, and a 32-bit
   seed for each of the feature points. This function is not
   difficult to extend to compute alternative information such as
   higher-order F values, to use the Manhattan distance metric, or
   other fun perversions.

   <at>    The input sample location.
   <max_order> Smaller values compute faster. < 5, read the book to extend it.
   <F>     The output values of F_1, F_2, .. F[n] in F[0], F[1], F[n-1]
   <delta> The output vector difference between the sample point and the n-th
            closest feature point. Thus, the feature point's location is the
            hit point minus this value. The DERIVATIVE of F is the unit
            normalized version of this vector.
   <ID>    The output 32-bit ID number that labels the feature point. This
            is useful for domain partitions, especially for coloring flagstone
            patterns.

   This implementation is tuned for speed in a way that any order > 5
   will likely have discontinuous artifacts in its computation of F5+.
   This can be fixed by increasing the internal points-per-cube
   density in the source code, at the expense of slower
   computation. The book lists the details of this tuning. */

#include <math.h>
#include <stdio.h>
#include <assert.h>
#include "cell.h" /* Function prototype */

/* A hardwired lookup table to quickly determine how many feature
   points should be in each spatial cube. We use a table so we don't
   need to make multiple slower tests. A random number indexed into
   this array will give an approximate Poisson distribution of mean
   density 2.5. Read the book for the long-winded explanation. */

static int Poisson_count[256]=

{4,3,1,1,1,2,4,2,2,2,5,1,0,2,1,2,2,0,4,3,2,1,2,1,3,2,2,4,2,2,5,1,2,3,
 2,2,2,2,2,3,2,4,2,5,3,2,2,2,5,3,3,5,2,1,3,3,4,4,2,3,0,4,2,2,2,1,3,2,
 2,2,3,3,3,1,2,0,2,1,1,2,2,2,2,5,3,2,3,2,3,2,2,1,0,2,1,1,2,1,2,2,1,3,
 4,2,2,2,5,4,2,4,2,2,5,4,3,2,2,5,4,3,3,3,5,2,2,2,2,2,3,1,1,4,2,1,3,3,
 4,3,2,4,3,3,3,4,5,1,4,2,4,3,1,2,3,5,3,2,1,3,1,3,3,3,2,3,1,5,5,4,2,2,
 4,1,3,4,1,5,3,3,5,3,4,3,2,2,1,1,1,1,1,2,4,5,4,5,4,2,1,5,1,1,2,3,3,3,
 2,5,2,3,3,2,0,2,1,1,4,2,1,3,2,1,2,2,3,2,5,5,3,4,5,5,2,4,4,5,3,2,2,2,
 1,4,2,3,3,4,2,5,4,2,4,2,2,2,4,5,3,2};
```

```
/* This constant is manipulated to make sure that the mean value of F[0]
   is 1.0. This makes an easy natural "scale" size of the cellular features. */
#define DENSITY_ADJUSTMENT 0.398150

/* the function to merge-sort a "cube" of samples into the current best-found
   list of values. */
static void AddSamples(long xi, long yi, long zi, long max_order,
                       double at[3], double *F,
                       double (*delta)[3], unsigned long *ID);

/* The main function! */
void Worley(double at[3], long max_order,
            double *F, double (*delta)[3], unsigned long *ID)
{
  double x2,y2,z2, mx2, my2, mz2;
  double new_at[3];
  long int_at[3], i;

  /* Initialize the F values to "huge" so they will be replaced by the
  first real sample tests. Note we'll be storing and comparing the
  SQUARED distance from the feature points to avoid lots of slow
  sqrt() calls. We'll use sqrt() only on the final answer. */
  for (i=0; i<max_order; i++) F[i]=999999.9;

  /* Make our own local copy, multiplying to make mean(F[0])==1.0 */
  new_at[0]=DENSITY_ADJUSTMENT*at[0];
  new_at[1]=DENSITY_ADJUSTMENT*at[1];
  new_at[2]=DENSITY_ADJUSTMENT*at[2];

  /* Find the integer cube holding the hit point */
  int_at[0]=(long)floor(new_at[0]); /* A macro could make this slightly faster */
  int_at[1]=(long)floor(new_at[1]);
  int_at[2]=(long)floor(new_at[2]);

  /* A simple way to compute the closest neighbors would be to test all
     boundary cubes exhaustively. This is simple with code like:
     {
       long ii, jj, kk;
       for (ii=-1; ii<=1; ii++) for (jj=-1; jj<=1; jj++) for (kk=-1; kk<=1; kk++)
       AddSamples(int_at[0]+ii,int_at[1]+jj,int_at[2]+kk,
       max_order, new_at, F, delta, ID);
     }
     But this wastes a lot of time working on cubes that are known to be
     too far away to matter! So we can use a more complex testing method
     that avoids this needless testing of distant cubes. This doubles the
     speed of the algorithm. */

  /* Test the central cube for closest point(s). */
  AddSamples(int_at[0], int_at[1], int_at[2], max_order, new_at, F, delta, ID);
```

Team LRN

```
/* We test if neighbor cubes are even POSSIBLE contributors by examining the
   combinations of the sum of the squared distances from the cube's lower
   or upper corners.*/
x2=new_at[0]-int_at[0];
y2=new_at[1]-int_at[1];
z2=new_at[2]-int_at[2];
mx2=(1.0-x2)*(1.0-x2);
my2=(1.0-y2)*(1.0-y2);
mz2=(1.0-z2)*(1.0-z2);
x2*=x2;
y2*=y2;
z2*=z2;

/* Test 6 facing neighbors of center cube. These are closest and most
   likely to have a close feature point. */
if (x2<F[max_order-1])  AddSamples(int_at[0]-1, int_at[1] , int_at[2] ,
                                   max_order, new_at, F, delta, ID);
if (y2<F[max_order-1])  AddSamples(int_at[0] , int_at[1]-1, int_at[2] ,
                                   max_order, new_at, F, delta, ID);
if (z2<F[max_order-1])  AddSamples(int_at[0] , int_at[1] , int_at[2]-1,
                                   max_order, new_at, F, delta, ID);

if (mx2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1] , int_at[2] ,
                                   max_order, new_at, F, delta, ID);
if (my2<F[max_order-1]) AddSamples(int_at[0] , int_at[1]+1, int_at[2] ,
                                   max_order, new_at, F, delta, ID);
if (mz2<F[max_order-1]) AddSamples(int_at[0] , int_at[1] , int_at[2]+1,
                                   max_order, new_at, F, delta, ID);

/* Test 12 "edge cube" neighbors if necessary. They're next closest. */
if ( x2+ y2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1]-1, int_at[2] ,
                                       max_order, new_at, F, delta, ID);
if ( x2+ z2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1] , int_at[2]-1,
                                       max_order, new_at, F, delta, ID);
if ( y2+ z2<F[max_order-1]) AddSamples(int_at[0] , int_at[1]-1, int_at[2]-1,
                                       max_order, new_at, F, delta, ID);
if (mx2+my2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1]+1, int_at[2] ,
                                       max_order, new_at, F, delta, ID);
if (mx2+mz2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1] , int_at[2]+1,
                                       max_order, new_at, F, delta, ID);
if (my2+mz2<F[max_order-1]) AddSamples(int_at[0] , int_at[1]+1, int_at[2]+1,
                                       max_order, new_at, F, delta, ID);
if ( x2+my2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1]+1, int_at[2] ,
                                       max_order, new_at, F, delta, ID);
if ( x2+mz2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1] , int_at[2]+1,
                                       max_order, new_at, F, delta, ID);
if ( y2+mz2<F[max_order-1]) AddSamples(int_at[0] , int_at[1]-1, int_at[2]+1,
                                       max_order, new_at, F, delta, ID);
if (mx2+ y2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1]-1, int_at[2] ,
                                       max_order, new_at, F, delta, ID);
```

Team LRN

```
  if (mx2+ z2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1] , int_at[2]-1,
                                         max_order, new_at, F, delta, ID);
  if (my2+ z2<F[max_order-1]) AddSamples(int_at[0] , int_at[1]+1, int_at[2]-1,
                                         max_order, new_at, F, delta, ID);

  /* Final 8 "corner" cubes */
  if ( x2+ y2+ z2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1]-1, int_at[2]-1,
                                         max_order, new_at, F, delta, ID);
  if ( x2+ y2+mz2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1]-1, int_at[2]+1,
                                         max_order, new_at, F, delta, ID);
  if ( x2+my2+ z2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1]+1, int_at[2]-1,
                                         max_order, new_at, F, delta, ID);
  if ( x2+my2+mz2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1]+1, int_at[2]+1,
                                         max_order, new_at, F, delta, ID);
  if (mx2+ y2+ z2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1]-1, int_at[2]-1,
                                         max_order, new_at, F, delta, ID);
  if (mx2+ y2+mz2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1]-1, int_at[2]+1,
                                         max_order, new_at, F, delta, ID);
  if (mx2+my2+ z2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1]+1, int_at[2]-1,
                                         max_order, new_at, F, delta, ID);
  if (mx2+my2+mz2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1]+1, int_at[2]+1,
                                         max_order, new_at, F, delta, ID);

  /* We're done! Convert everything to right size scale */
  for (i=0; i<max_order; i++)
    {
      F[i]=sqrt(F[i])*(1.0/DENSITY_ADJUSTMENT);
      delta[i][0]*=(1.0/DENSITY_ADJUSTMENT);
      delta[i][1]*=(1.0/DENSITY_ADJUSTMENT);
      delta[i][2]*=(1.0/DENSITY_ADJUSTMENT);
    }

  return;
}

  static void AddSamples(long xi, long yi, long zi, long max_order,
                         double at[3], double *F,
                         double (*delta)[3], unsigned long *ID)
  {
  double dx, dy, dz, fx, fy, fz, d2;
  long count, i, j, index;
  unsigned long seed, this_id;

  /* Each cube has a random number seed based on the cube's ID number.
     The seed might be better if it were a nonlinear hash like Perlin uses
     for noise, but we do very well with this faster simple one.
     Our LCG uses Knuth-approved constants for maximal periods. */
  seed=702395077*xi + 915488749*yi + 2120969693*zi;

  /* How many feature points are in this cube? */
```

```
count=Poisson_count[seed>>24]; /* 256 element lookup table. Use MSB */

seed=1402024253*seed+586950981; /* churn the seed with good Knuth LCG */

for (j=0; j<count; j++) /* test and insert each point into our solution */
  {
   this_id=seed;
   seed=1402024253*seed+586950981; /* churn */

    /* compute the 0 .. 1 feature point location's XYZ */
    fx=(seed+0.5)*(1.0/4294967296.0);
    seed=1402024253*seed+586950981; /* churn */
    fy=(seed+0.5)*(1.0/4294967296.0);
    seed=1402024253*seed+586950981; /* churn */
    fz=(seed+0.5)*(1.0/4294967296.0);
    seed=1402024253*seed+586950981; /* churn */

    /* delta from feature point to sample location */
    dx=xi+fx-at[0];
    dy=yi+fy-at[1];
    dz=zi+fz-at[2];

    /* Distance computation! Lots of interesting variations are
       possible here!
       Biased "stretched" A*dx*dx+B*dy*dy+C*dz*dz
       Manhattan distance fabs(dx)+fabs(dy)+fabs(dz)
       Radial Manhattan:  A*fabs(dR)+B*fabs(dTheta)+C*dz
       Superquadratic:    pow(fabs(dx), A) + pow(fabs(dy), B) + pow(fabs(dz),C)

       Go ahead and make your own! Remember that you must insure that a
       new distance function causes large deltas in 3D space to map into
       large deltas in your distance function, so our 3D search can find
       them! [Alternatively, change the search algorithm for your special
       cases.]
    */

    d2=dx*dx+dy*dy+dz*dz; /* Euclidean distance, squared */

    if (d2<F[max_order-1]) /* Is this point close enough to rememember? */
      {
    /* Insert the information into the output arrays if it's close enough.
       We use an insertion sort. No need for a binary search to find
       the appropriate index .. usually we're dealing with order 2,3,4 so
       we can just go through the list. If you were computing order 50
       (wow!!), you could get a speedup with a binary search in the sorted
       F[] list. */

    index=max_order;
    while (index>0 && d2<F[index-1]) index--;
```

```
      /* We insert this new point into slot # <index> */

      /* Bump down more distant information to make room for this new point. */
      for (i=max_order-1; i-->index;)
         {
           F[i+1]=F[i];
           ID[i+1]=ID[i];
           delta[i+1][0]=delta[i][0];
           delta[i+1][1]=delta[i][1];
           delta[i+1][2]=delta[i][2];
         }
      /* Insert the new point's information into the list. */
      F[index]=d2;
      ID[index]=this_id;
      delta[index][0]=dx;
      delta[index][1]=dy;
      delta[index][2]=dz;
    }
  }

 return;
}
```

```
      /* We insert this new point into slot # <index> */

      /* Bump down more distant information to make room for this new point. */
      for (i=max_order-1; i-->index;)
         {
           F[i+1]=F[i];
           ID[i+1]=ID[i];
           delta[i+1][0]=delta[i][0];
           delta[i+1][1]=delta[i][1];
           delta[i+1][2]=delta[i][2];
         }
      /* Insert the new point's information into the list. */
      F[index]=d2;
      ID[index]=this_id;
      delta[index][0]=dx;
      delta[index][1]=dy;
      delta[index][2]=dz;
    }
  }

 return;
}
```

5

Team LRN

ADVANCED ANTIALIASING

STEVEN WORLEY

It's very tempting to ignore the problem of antialiasing when writing textures. Certainly, the first textures anyone writes are very quick tools that make you happy to get any kind of pattern at all on your objects. But as you mature, you learn that even the simplest textures behave very poorly because of the problem of aliasing. This is why the topic of antialiasing is discussed so avidly in so many places in this book.

Aliasing has different definitions depending on context, but it ultimately resolves to the problem that you want to show the *average* effect of a texture over an area, yet it's a lot easier to just return a *sample* of the texture at just one infinitely small point. If you ignore the problem of antialiasing, you'll start making imagery that simply looks bad. You'll find artifacts like stippling or stairstepping in your images. You'll have especially visible problems when you make animations, since the motion of objects tends to highlight any aliasing problem by causing buzzing in the image—or worse.

It's very tempting to try to ignore the problem anyway, especially since most renderers have options to perform supersampling of the image for you to automatically reduce the aliasing problems. But don't depend on this! Your users may not need the whole image supersampled if the only problem is your texture. Image supersampling is expensive. *Supersampling is not an answer to the problem of texture aliasing.* It's just a final brute-force attempt to hide the problem.

The obvious objection to adding antialiasing abilities to textures is efficiency. Textures tend to be slow, and adding any more baggage (such as built-in integration for antialiasing) is bound to slow them even more, as well as increase the complexity of the code. This seems to argue for shorter, dumber textures, but in practice this is not true; a texture that can antialias itself can do so with a single (albeit slower) evaluation. Supersampling might take a dozen samples and still be less accurate. Correct antialiasing can therefore be an efficiency issue—and an important one.

Antialiasing is unfortunately a lot of work for the texture designer, since it often requires careful thought. It often takes a different design method to do proper

Team LRN

texture area integration, and the new method is never easier than simple point sampling.

This chapter covers only some aspects of antialiasing and in particular does not discuss band limiting, covered by Darwyn Peachey in Chapter 2.

INDEX ALIASING

It can be useful to look at the sources of aliasing in order to identify any components that we might be able to improve in behavior. In particular, nearly all textures can be summarized as being some formula or procedure (call it the "pattern" function) that returns a scalar defined over space. This value is transformed into a color or other attribute using a second function. This final transformation function might be something as simple as a linear gradient, but it can be abstracted to be a general lookup table that can characterize any behavior at all.[1] This design paradigm is convenient because it's not difficult to implement and is very versatile for the user.[2]

If we have a texture that follows this design method, we can see that aliasing is not caused from a single source, but from two. Imagine a point color sample being computed. The scalar pattern function is called, which returns a number. This number is used to index into the color transformation (lookup table), which identifies the color to return.

There is a source of aliasing in both steps of this process. Point-sampling the scalar pattern function obviously cannot characterize the average behavior of the pattern over the area. Less obviously, aliasing is caused in the color transformation step. A single value is used to determine the output color. This causes aliasing too, although it's more difficult to identify.

To illustrate a worst-case example, imagine our pattern function is a single scale of Perlin-style noise, and it varies between roughly −1 and 1. We choose a color map that is green for all input values except a very narrow band of bright red centered at the value corresponding to 0. This would make a pattern that is primarily green everywhere except for tiny spots of red where the noise value happens to be right at

1. A caveat here: since a table has only a finite number of entries, of course it can't perfectly represent any transformation. Luckily, in practice most of these transformations are simple enough that using just a few hundred entries does an excellent job of "summarizing" the transformation. Using a few thousand table entries is very likely to be adequate for any task. This is just a couple K of storage, so it's not a big overhead.

2. Note that there is more discussion of this methodology on pages 181 and 182.

Team LRN

0.0. Now, if we try to antialias this texture, we run into a large problem if our integration spot size is very large compared to the variation scale of the noise function. The average value of the noise over the large integration spot will converge to 0. Yet if we feed this perfectly integrated value into our color lookup table, we get a solid red color even though the true average color is a mostly green shade! This is a worst-case demonstration of aliasing in the final color transformation stage, sometimes termed *index aliasing*.

This aliasing is caused by rapid changes of the function used to transform scalar pattern values into output colors. Another example can help show how this aliasing might occur. Figure 5.1 shows a nontrivial color map that might be used to transform a fractal noise value into one component of the surface color. If we evaluate the fractal noise function several times, we get several corresponding color point evaluations. It can be seen that in this example, the samples lie in one region of the color map; does the average of the point samples accurately reflect the average of the region? We can convince ourselves that other noise samples are likely to occur through

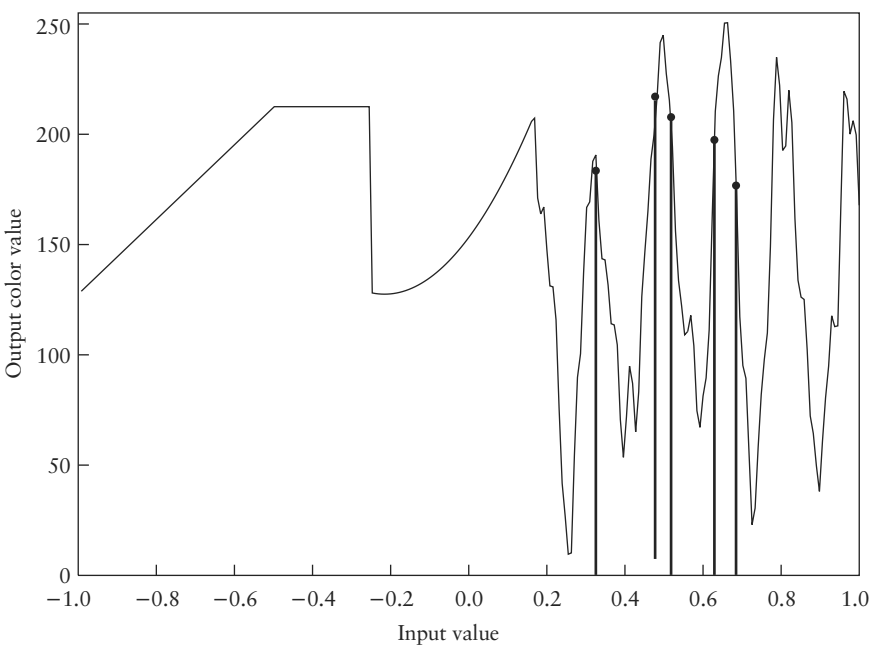


FIGURE 5.1 Point samplings of a complex color map.

Team LRN

the region from 0.3 to 0.7 and that the set of five current samples, by simple bad luck, probably shows a color value that is higher than the true mean color over this region.

With some thought, we can design a strategy to significantly reduce this source of aliasing. To do this, we need to think about what the true output should be and how it is created. Then we can examine our approximations and see how we might modify our strategy. We'll consider the scalar "pattern" function as a black-box function so the method will be applicable to any texture that uses a color map in this way.

The true output of an antialiased texture cannot be described by a single sample of our color spline. This is quickly seen by our red/green color example, since the average integrated color doesn't even appear in the color spline. If we consider what the average of an infinite number of point texture and color samples would converge to, we'll recognize the fact that the final average color is a weighted sum of the color spline. These weights are determined by the distribution of the pattern function's samples over the area in question. We can immediately see that since the *distribution* is what is important, a single sample value (like one evaluated at the mean of the distribution) is inadequate for determining the final average color.

What is the distribution defined by the pattern function over the integration area? *We don't know,* since we're treating the pattern function as a black box. We can make at least one assumption: the distribution is probably roughly continuous. We can make this assumption because most useful pattern functions (think of fractal noise, or a field that changes linearly with distance from a point or line, or a perturbed sine wave value) do not have discontinuities. Sharp changes *tend* to occur in the color transformation step, not the pattern function. If the pattern function is continuous, the samples taken over a local area will tend to cover a local range of values. This does not mean that all of these covered values will be equally likely, just that it's unlikely to have multiple separate peaks in the distribution with no samples in between.

Our only method of exploring the pattern function's distribution is to take point samples. We could implement naive supersampling by just indexing each of these samples into the color lookup table and averaging the output colors. But we can do better by thinking about the problem a bit more, perhaps with a simple example. Imagine we take just two samples of the pattern function. What is our best estimate for the final surface color?

Here we use our assumption of continuity. We can make an admittedly vague guess that the samples we took imply that future samples will occur between the two samples we have. Without any more information, a valid argument is that we can

Team LRN

make a best guess of the distribution of the pattern value by simply assuming that all values between the two samples are equally likely.

If we have a guess like this for the distribution of the pattern values, how can we turn this into a best guess for the final color output? For a certain input distribution, we can compute the final color distribution and therefore the mean of this output. Since we're modeling our input distribution as a box of equally likely values between the samples we've taken, we can simply integrate the color lookup table between these values to find the mean. The trick to try to eliminate the index aliasing is to try to *model the input distribution as best we can,* since then we can integrate the (explicitly known) color map using that distribution as a weighting to get as accurate an estimate of the average color as possible.

Two important questions remain: How do we best model that input distribution? And how do we compute the weighted integration of the color map efficiently? The first problem must be solved by sampling the black-box pattern function. For two samples, an argument can be made for using a uniform distribution between the two sample values. How about for three samples? If values of 1.0, 1.1, and 1.5 are returned, what is our best guess for the distribution? With such limited information, we can somewhat visualize a range of values between 1.0 and 1.5, with a lopsided distribution toward the low values. Yes, this might not be what the true distribution looks like, but it is our best guess.

Such a vague definition of a distribution is inadequate, and especially when higher numbers of samples are known, we must have a general method of computing an approximation to the input distribution. The best method seems to be the following: If we take N samples, we make the assumption that if we sort the values in ascending order, new samples are equally likely to occur between each adjacent pair of samples. This models a piecewise flat distribution, much like a bar graph. Figure 5.2 shows a collection of point samples from some (unknown) distribution, as well as a best-guess reconstruction of the distribution function. Each "bar" that forms the

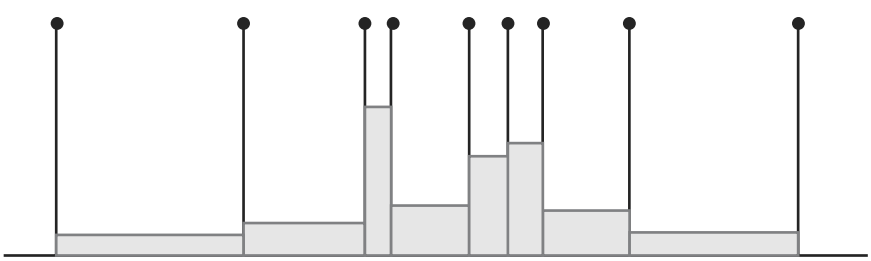


FIGURE 5.2 Reconstruction of an unknown distribution from point samples.

Team LRN

distribution denotes an equal probability, and therefore each has the same area. Note that this transforms a high density of samples (which are spaced closely together) into tall, high-probability regions.

This method for reconstructing a "best-guess" distribution has three advantages. First, it behaves as we would expect for small numbers of samples: one sample isn't enough to guess a distribution, two make a simple bar, and our example with three points does compensate for the lopsidedness of the sample points. Second, the reconstruction behaves better and better as more points are used. It will converge to the true distribution, even those with discontinuities or gaps. Third, we'll see that a piecewise constant distribution helps us perform the second required task for index antialiasing, which is to weight the color lookup table by the distribution and efficiently find the mean output color.

To do this integration, we can first develop a method of integrating a lookup table quickly over a range of values with uniform weight. This is equivalent to finding the sum of the table entries over the range, then dividing by the number of entries summed. There is an easy method for performing sums like this, called a *sum table*. These are in fact already used in computer graphics, primarily for 2D image map antialiasing. Here we wish to perform the sum on a mere 1D function, which is even easier.

A sum table is a simple concept. If we have an array of values and want to be able to sum those values over a certain interval, we can use a second, precomputed array called the sum table. Each entry of the sum table is equal to the sum of the entries of the original table up to that entry number. If our original table T were [1 3 4 2 3], our sum table S would be [1 4 8 10 13]. Now if we want to sum the values of entries a to b in T inclusively, we can simply evaluate $S(b) - S(a - 1)$.[3]

Thus, by two simple table lookups and one subtraction, it is an easy matter to sum any number of entries in the original table. Looking at our reconstructed input distribution, the piecewise uniform weights lend themselves to this summed table evaluation perfectly. Each interval of the distribution is assigned a weight (inversely proportional to its width). The mean value of the color table over the interval is found through the difference of two sum table entries divided by the number of entries spanned. This mean is weighted by the region's weight, and the complete weighted color sum is divided by the sum of all the weights to determine the final output color estimate.

Since the end of one interval is shared by the start of the next, we can even combine the terms containing the same indexing into the sum table. When we have N

3. A bit of care has to be taken when $a = 1$, where we assume $S(0)$ is 0.

Team LRN

samples of the pattern function, we only need N indexes into the sum table. This makes the evaluation extremely economical.

In some cases you can simplify the calculation further by calculating the mean and standard deviation of the samples. You can average the texture value over the color table range defined by the central mean value with total width equal to twice the standard deviation. This method simplifies the color map calculation to two lookups and also does not require sorting.

Index antialiasing is just one step of full texture antialiasing, but it's an extremely useful one. It's much more effective than blind supersampling, since (especially for low numbers of samples) at least a reasonable model of the variation of the texture is used to estimate the variation the color map undergoes. Implementation of index antialiasing is very easy, although it does assume a lookup table for your color map. This table will need to be filled, taking some preprocessing time and extra memory. This overhead is very small, however, even for a very large table, since the tables are only one-dimensional.

There are several extensions and adaptations of this algorithm, especially if you know something more about your pattern function. Sometimes you might be able to make assumptions about the shape of the sample distribution or be able to compute the true distribution analytically. For example, if the pattern function is defined by the distance of a point from a line, then the spot size defines a circle[4] of points. The true distribution of the sample values can also be determined algebraically or even geometrically. This can allow you to perform nearly perfect antialiasing! Obviously, the more you know about your pattern function, the better you'll be able to model its sample distribution over an area, and the better your antialiasing will become.

An Example: Antialiasing Planetary Rings

A great example of index aliasing is in generating planetary rings. The fine ring structure is a function of radius, but often the surface is viewed from far away and a single sample may range over a large range of radii, and many rings should be "averaged" together to provide a filtered version of the ring effect.

The planetary ring texture here uses the index aliasing reduction technique to provide a smoothly filtered version of the complex rings efficiently.

```
/* this is a 1D table, we can splurge for lots of entries and it still
   won't be too large to store. */
```

4. An ellipse, actually.

```
#define TABLESIZE 16384

int initialized=0;
float table[TABLESIZE];

static double RingTexture(double *pos, double spotsize,
  double inner, double outer, double transition,
  double varyscale, double ampratio,
  double low, double high)
{
  double R0, R1, weight;
  int I0, I1;

  /* We need a color table premade for us with the accumulated densities. */
  if (!initialized) MakeColorTable(inner, outer, transition,
  varyscale, ampratio, low, high);

  /* OK, let's find our range of radii. The range of radii are centered
     at the point's radius from the origin, and we split the spot radius
     between the two.
     To tweak the antialiasing, we could scale the spotsize up or down
     here before we use it.
     */

  R0=sqrt(pos[0]*pos[0]+pos[1]*pos[1]+pos[2]*pos[2])-0.5*spotsize;
  R1=R0+spotsize;

  /* R0 may be negative if we have huge spot sizes, which is obviously
     wrong and awkward. Let's make sure it's at least 0.0. */

  R0=MAX(R0, 0.0);

  /* OK, we know what range we want to average over. Let's transform those
     into index numbers in our color table. We know that at radius = [outer],
     we want to be at index [TABLESIZE-1]. So we multiply by a
     constant.

     Note that we lose a little resolution because we quantize the
     index into an integer. If the table were smaller, we could
     compensate by using a linear interpolation, but with a really big
     table the effect is negligible, and it definitely makes the code simpler.

  */

  I0=(int)(R0*(TABLESIZE-1)/outer);
  I1=(int)(R1*(TABLESIZE-1)/outer);
  /* We make a tiny change to make sure I0 and I1 are not coincident, which
     simplifies tests later. This case will rarely happen anyway. */
  if (I0==I1) I1++;
```

```
  /* These indexes may be out of range. If so, let's do the right thing. */
  if (I0>=TABLESIZE) return 0.0; /* We're completely outside the rings,
    no effect. */
  if (I1>=TABLESIZE)
    {
     /* Our outer range has run "off" of the end of the result. We have to
take this into account by figuring the ratio of what's fallen off
to what remains, and make sure our final output is properly
weighted. */

     weight=((double)(TABLESIZE-I0))/(I1-I0);
     /* Now we change I1 to start within the range. The weight parameter
will compensate for the range change. */
     I1=TABLESIZE-1;
    }
  else weight=1.0; /* We're fully within the rings, we'll use the full
     weight. */

  /* We now want the average value between I0 and I1. This is the
     easy part! */

  return weight*(table[I1]-table[I0])/(I1-I0);
}

/* Routine to build the summed color table itself. This includes the
   computation of the ring density tucked within a simple loop to
   build the sum table as it goes. This is a precomputation that only
   needs to be done once. */

static void MakeColorTable(double inner, double outer, double transition,
    double varyscale, double ampratio,
    double low, double high)
{
  double R, A, F;
  int i;

  /* Sweep the radii out to the outer radius. Accumulate samples in
     the summed color table. Point-sample the ring density at each sample.
     A radius of 0 is index 0. A radius of [outer] is equal to the table
     size-1.
  */

  table[0]=0.0; /* Start with a 0 table */

  for (i=0; i<TABLESIZE-1; i++)
    {
     R=outer*(i+0.5)/(TABLESIZE-1); /* R varies between 0 and [outer] */

     /* Compute the simple inner/outer transitions to form an alpha channel
 of a simple washerlike disk. This will be used to modulate the
```

```
 density of the fine rings and prevent any rings from being too
 close or too far from the center. */

     if (R<=inner) A=0.0;
     else /* In first transition zone? */
if (R<inner+transition)
  {
    A=(R-inner)/transition; /* Linear 0 to 1 ramp */
    A*=A*(3.0-2.0*A); /* Hermite curve smooth transition */
  }
else /* In outer transition zone? */
  if (R>outer-transition)
  {
    A=(outer-R)/transition; /* Linear 1 to 0 ramp */
    A*=A*(3.0-2.0*A); /* Hermite curve smooth transition */
  }
  else A=1.0; /* We're in the main body of the ring. */

    /* Now let's compute the ring density. We use a 1D version of
 fractal noise. We use a 3D noise routine but pass in (R, 0, 0)*/

    F=fractal3(R, 0.0, 0.0, varyscale, ampratio);

    /* F is now between -1 and 1. But we use the low and high values as
 a clipping range for the noise. */

    if (F<=low) F=0.0; /* F is too low, we set the ring density to 0.0. */
    else if (F>=high) F=1.0; /* Full ring density */
    else /* We're in a transition zone. */
{
 F=(F-low)/(high-low); /* Now a 0 to 1 range */
 F*=F*(3.0-2.0*F); /* Hermite it to make it smooth */
}

    /* OK, our ring density is F and our shaping alpha value is A. The
 net density is the PRODUCT of these two. */

    table[i+1]=table[i]+R*F;
    }
    initialized=1;
    return;
}
```

SPOT GEOMETRY

Your antialiasing goal is usually to find the *average* texture value over a small area. This area is known as the *spot size*, and usually the renderer will tell you what this size is. Some renderers like RenderMan are very careful in computing these spot

sizes, but others (especially ray tracers, it seems) are very careless and give no spot size or (perhaps worse) an incorrect spot size.

This spot size is easy to understand if you think about a very simple renderer that takes one sample per pixel and renders a simple plane with a texture on it. Since it's rendering with just one sample per pixel, the texture algorithm will ideally return the average color of the texture pattern over that pixel. But textures don't know or care about pixels; they almost always want to be given a *location* in texture coordinates. What you usually want is a texture center location, in *XYZ* coordinates, and a radius (the spot size) measured in that same coordinate system.

If you're lucky, the renderer will give you this radius at the same time as the sample location. For example, RenderMan's rendering method chops objects into smaller and smaller parts until each bit is smaller than a pixel when it is projected onto the output image. During this slicing and dicing, it keeps track of the exact range of texture coordinates for each little bit, known as a micropolygon. It ultimately reduces surfaces to a tiny rectangular chunk of texture with exactly known texture coordinate ranges, which it passes to the texture to be evaluated. RenderMan is extremely texture-friendly because of this careful coordinate treatment.

More conventional renderers usually concern themselves with projecting the geometry onto the screen and then, grudgingly, determining the texture coordinates to pass to the texture code. These tend to be point locations computed by starting with a world coordinate value and transforming first to object and then to texture coordinates. The spot size is then ideally computed by analyzing the transformation between texture space and image space and using this to approximate a texture spot size.

If the previous sentence sounds like vague hand-waving, that's because *very few renderers compute spot sizes very well.* They all use different techniques and approximations. I've had personal experience with three different renderers, and each had inaccurate spot sizes. There are several ways of dealing with this inaccuracy, even if you have no control over the renderer itself. One method is to find correction factors to "massage" the spot size back to be more accurate by using different constants to scale the spot size.[5]

Another method is to compute the spot sizes yourself, if you have enough information. The derivation is straightforward but involves several steps and simplifying assumptions. It can be frustrating to locate all the information needed to complete each step, since renderers often don't give you all of the surface data you need.

5. This is straightforward to do with the image-based verification method described on page 174.

Team LRN

First, we must know what size the spot we are antialiasing is in the *output image*. This is often the size of one pixel, but in the case of image supersampling the size may be smaller.

We first need a conversion to change a world coordinate into an image coordinate *I*. (This is the formula that takes any 3D location in world coordinates and predicts where the point will project onto your output image.) This relation is usually computed by two transformations. The first is a rotation and offset transformation to translate the camera to a coordinate system where it is at 0, 0, 0 and looking forward. The second transformation is the perspective transform, which projects the (transformed) 3D point onto the screen. They are often combined into a single operation, although it may be easier to think of them as two sequential operations.

We then need a transformation from world coordinates to texture coordinates. This is usually a simple linear matrix equation with an offset. It is often the concatenation of two transformations, the first from world to object space and the second from object to texture space.

By concatenating the effects of each transformation in the sequence together, we arrive at a nonlinear transformation function, $I(T)$, which relates texture coordinates to screen coordinates. Its exact representation depends on your camera model and coordinate system definitions, so even its form tends to be unique to every renderer.

Next, we make an assumption that the texture area we are averaging over is so small that it can be well approximated by a planar sample through the texture. This is not always true (think of a sphere that is so distant that it is a single pixel in size), but the assumption holds well in almost every case.

If the antialiasing spot we are averaging over is a flat 2D area, the direction perpendicular to this region must be the same direction as the surface normal (in texture coordinates). The surface normal in fact defines the plane's normal. We can pick two vectors (call them *R* and *S*) that are mutually perpendicular to each other and also to the surface normal *N*. These three vectors define a new coordinate system. We know that the texture spot is defined only in the plane of *R* and *S*.

We can choose which *R* and *S* to use by thinking about the geometry of the texture spot. If the surface normal *N* directly faces the camera, the spot will be a flat circle, and we can use any *R* and *S* that are perpendicular to *N* since there's no preferred spot direction.

If the surface normal does not directly face the camera, the texture spot will be tilted slightly away from us. This will make the texture spot *elongated* in one direction. If the view direction toward the camera is *V*, the elongated texture direction will lie in the direction of *V* projected onto the surface.

We want to align our texture spot in this direction in order to simplify our antialiasing later. In practice, we often do this in world coordinates first. We start

Team LRN

with V_w, the world coordinate view direction from the texture location toward the camera, and N_w, the surface normal in world coordinates. We form S_w as the *non-elongated* direction by taking the cross product $V_w \times N_w$ and normalizing. R_w, the direction along the elongation, is computed by normalizing $S_w \times N_w$. We then transform S_w and R_w into texture coordinates by using the world-to-object and object-to-texture transformations.

If the texture sample position (which defines the center of the area we want to average over) is T, we can parameterize the region we want to average over as $T + rR + sS$, where r and s are scalars. But we don't know what range of r and s should be used—yet. To determine them, we can use our formula we computed earlier for transforming from texture coordinates to screen coordinates. If we substitute our parametric spot equation into the projected image formula, $I(T + rR + sS)$ tells us where on the screen a texture sample at parametric coordinate (r, s) falls.

We can now differentiate I with respect to r and s. We do this twice, since I is really a coordinate pair (x, y). This gives us $\frac{dI_x}{dr}$, $\frac{dI_y}{dr}$, $\frac{dI_x}{ds}$, and $\frac{dI_y}{ds}$.

We can stop here and return an isotropic spot size that gives just a single radius to average over by noting that if we want to average over a pixel, we want to move a small delta of 0.5 pixels up, down, left, and right in the image. We can use something similar to

$$\Delta = \sqrt{\left(\frac{dI_x}{dr}\right)^2 + \left(\frac{dI_y}{dr}\right)^2 + \left(\frac{dI_x}{ds}\right)^2 + \left(\frac{dI_y}{ds}\right)^2}$$

to get a kind of "average spot radius." This result isn't bad, and most renderers stop here and give a value similar to this. We can stop here if we want and just integrate our texture over $(-\Delta .. \Delta, -\Delta .. \Delta)$.

But we've nearly derived a much higher-quality spot computation that accounts for stretched spots. We know how I changes with both s and r. We can set up two linear equations,

$$\left(\frac{dI_x}{dr}\right)\Delta r + \left(\frac{dI_x}{ds}\right)\Delta s = 0.5$$

$$\left(\frac{dI_y}{dr}\right)\Delta r + \left(\frac{dI_y}{ds}\right)\Delta s = 0.5$$

which states that we assume our image spot to vary half a pixel in both x and y, and we expect r and s to vary by $\pm \Delta r$, $\pm \Delta s$ to allow the texture spot to vary over the entire pixel. The solution of these equations is

Team LRN

$$\Delta s = \frac{\frac{dI_x}{dr} - \frac{dI_y}{dr}}{2\left(\frac{dI_x}{dr}\frac{dI_y}{ds} - \frac{dI_x}{ds}\frac{dI_y}{dr}\right)}$$

and

$$\Delta r = \frac{\frac{dI_x}{dr} - \frac{dI_y}{dr}}{2\left(\frac{dI_x}{dr}\frac{dI_y}{ds} - \frac{dI_x}{ds}\frac{dI_y}{dr}\right)}$$

These give us the ranges $(-\Delta r, \Delta r)$ and $(-\Delta s, \Delta s)$ that multiply the R and S direction vectors to define the texture antialiasing geometry. You can view this as an ellipse or rectangle in texture space. This is a great spot geometry because we can antialias over an elongated spot if we care to, and we can also change our spot size if we know more about the shape of the sample in image space.

This computation seems somewhat daunting, with multiple transformations to compute on the fly and terrible math equations to solve. In practice, it's not quite so bad because the transformation formula $I(T)$ doesn't change from texture sample to sample. The work that must be done does involve several multiplies and divides, but most of the overhead comes from the square roots used in normalizing the S and R vectors.

With a computation like this, it's easy to see how a renderer can give poor spot size estimates if it is careless.

SAMPLING AND BUMPING

Some textures can be analytically integrated, like step functions and thick lines. Other textures have certain behaviors that can give at least better approximations than point sampling, such as band-limiting fractal noise scales to match the sample size. However, really odd textures just aren't practical to modify or even understand because of their custom design or complexity.

Supersampling is a last resort to reduce aliasing artifacts. It always works. It's usually inefficient and crude, but if we're forced to do this supersampling, we can at least do it intelligently.

We can benefit from having the *texture* perform its own antialiasing (even by supersampling) instead of the renderer. A short header at the beginning of the texture does its own supersampling of the surface area and returns the mean result. This is useful for several reasons. First, this means that all textures are treated the same by the renderer; a position and spot radius are passed, and the texture returns an

Team LRN

integrated texture estimate. The renderer does not need to treat antialiasing textures differently than ones that can only be point-sampled. Second, since the evaluation loop for the samples is within the texture, not the renderer, overhead (in particular, function calls) are reduced. This is a small savings in general, but for simple textures like checkerboards it can be significant in proportion to the texture's overall speed.

But how should a texture antialias itself with point samples? How does it pick the locations? Luckily, we already know the answer! The previous section on "Spot Geometry" has told us exactly how to vary our samples to cover the range of the texture's spot.[6]

In the simplest case, we just need to evaluate the texture multiple times over the range of spots defined by Δr and Δs in our texture coordinates. The average response of these samples is an antialiased supersample of the texture. But if you've read the "Index Antialiasing" section, you'll realize that technique is designed for this kind of point supersampling.

But there's a further bonus that is very easy to miss. A very common technique of bump mapping is to compute a vector derivative of a function and "perturb the surface normal" with it. But many times, this perturbation is done incorrectly because it often looks good anyway! In particular, if you take the derivative of a function (for example, fractal noise) and simply add it to the shading normal and renormalize, you'll get good-looking bumps, as shown in Figure 5.3(a). This is exactly what Ken Perlin did in his 1985 paper with its fabulous images. It's also what I did for dozens of commercial procedural textures. *But it's wrong!*

The right answer is to use the bump-mapping formula first shown by Blinn (1978), which Darwyn Peachey discusses in Chapter 2.[7] It's easy to skip this because the math is annoying. But, if you look at Figure 5.3, you'll see why understanding the texture spot geometry is so important.[8]

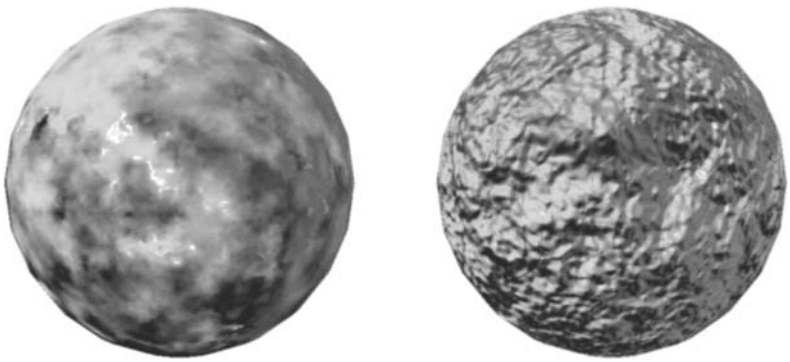
Why does the "simple" method work? Because it looks good and especially for fractal noise patterns you don't have any "correct" sample for your eyes to compare it to. There are clues, however, that indicate that it's wrong, especially if you animate

6. I know you skipped that section because it didn't seem exciting and had some ugly-looking math formulas. You're reading *this* section first because of the bumpy sphere pictures. You can improve your textures too, once you understand your spot geometry, though . . .

7. And again, a very good renderer will do all of this for you. This is one of the simple secrets of why RenderMan procedural textures tend to look so good! But many other renderers, even high-end commercial ones, do *not* do this for you!

8. I wish I had understood this for my 1992 commercial texture collections!

Team LRN



(a) (b)

FIGURE 5.3 Incorrect bumping of many textures: (a) "classic" bumping versus (b) correct.

your object rotating; the lighting will simply be inconsistent because the normal is incorrect.

When we know the spot geometry, we have all of the information we need to do "correct" bumping. If we're doing a small supersampling of our texture (to help eliminate aliasing), we can use the same engine to compute the bumping for us simultaneously. We don't even need to evaluate the derivative of our texture, since our point samples can tell us the average derivative over our spot, which is all that matters.

If we're sampling over our texture spot, we might have part of our texture return a single scalar value for the bump height of the surface. If we evaluate the height over the whole spot, we can fit an average "tilt" to the texture spot. We then just tilt our surface normal to match the tilt of the texture spot, and we get the correct surface bumping.

Luckily, computing this tilt is easy! It's just a least-squares fit of the arbitrary texture height function $H(r, s)$ to a line, repeated for the R and S directions. If we take samples symmetrically around the center of the spot, the slopes of the tilts are easy to compute:[9]

$$\frac{dH}{dr} = -\frac{\sum rH(r,s)}{\sum r^2}$$

9. Using a 1D least-squares line fitting method, which isn't perfect for super-steep slopes but is more than good enough.

and

$$\frac{dH}{ds} = -\frac{\sum sH(r,s)}{\sum s^2}$$

We then add these slopes to the surface normal. We need to do this in *world coordinates* since that's probably what the renderer expects. But this is just

$$N_{\text{bumped}} = \frac{N_{\text{original}} + \frac{dH}{dr} R_w + \frac{dH}{ds} S_w}{\sqrt{\left(1 + \left(\frac{dH}{dr}\right)^2 + \left(\frac{dH}{ds}\right)^2\right)}}$$

The √ term is just to renormalize the surface normal. Note that we are using the world coordinates R_w, S_w instead of texture coordinate R, S.

As a practical example, imagine sampling our texture at four texture locations, at the corners of our texture spot. We have some height function (maybe a fractal noise value) $H(r, s)$. So we can expand out the equations and find

$$\frac{dH}{dr} = \frac{\Delta r(H(-\Delta r, \Delta s) + H(-\Delta r, -\Delta s) - H(\Delta r, \Delta s) - H(\Delta r, -\Delta s)}{4(\Delta r)^2}$$

and similarly for S. If you look closely, you can even see how this is a finite-difference approximation to the derivative. But it's better than knowing even the exact true derivative at the center of the spot, because this is an *average* derivative over the whole spot and it will not suffer nearly as much aliasing.

This entire section on spot geometry is really a topic that authors of renderers should understand and implement, but unfortunately they often don't. Luckily, we texture authors can sometimes do the work ourselves to get good results. The proper texture spot definitions can give better antialiasing, proper sampling, and great bump mapping.

OPTIMIZATION AND VERIFICATION

After you've added antialiasing support or new spot geometry computations, it can be difficult to determine exactly how well it's working. The most common method is to render an image, zoom into the detail, see that it looks kind of blurry, and congratulate yourself.

This obviously isn't very scientific! In fact, it's hard to judge by eye whether your antialiasing is adequate or not. You can argue that if your eye can't tell, it doesn't matter, but in practice it's good to minimize it even below visible levels, since a

Team LRN

different application (perhaps with more extreme conditions) may amplify even a small amount of aliasing artifacts into a real problem.

There's a surprisingly straightforward and useful method for tweaking antialiasing for optimum results. It is important to reduce your measure of antialiasing "goodness" into a measurable number that you can actually try to optimize.

This can be done in nearly any situation by forming a controlled scientific experiment. It's usually very easy to adapt the following procedure to optimize antialiasing in every case.

First, you need to design a scene to render that exercises your texture. Strive to include situations that will typically cause problems. You want to have a single image that shows your texture at different scale sizes (infinite planes work well for this) as well as different viewing angles (spheres work well for this). I often use four infinite planes forming a box, receding to infinity, with about 10 spheres at different depths.

You need to generate a "reference" image of this scene, one that is as accurate as possible, including texture antialiasing. This can be done by rendering your scene at very high resolution, such as 4K by 4K, then filtering the image down in size to something more manageable like 256 by 256. This "manual supersampling" is nearly foolproof in making a good antialiased image since so many samples are used per pixel. Be wary of letting your renderer do supersampling for you when building a reference image! Its own supersampling may be biased or have subtle errors of its own.

This reference image provides a measure that we can compare our antialiasing against. When we render with no image supersampling, we can detect errors due to aliasing by simply comparing the rendered image with the reference image, pixel by pixel. The match is far from perfect because the reference image also includes *geometric* antialiasing, but this doesn't upset our texture antialiasing comparison.

The comparison between our test render image and the reference image should be done numerically. The most obvious error metric is to use a simple sum of the squared differences for each pixel. If the images are identical, this value will be 0. You can write a small application that takes the difference between two images and returns the *numeric* error value. It can also be interesting to output an image that highlights the pixels that have the most error.

With a tool for determining the antialiasing error, it becomes very easy (and, surprisingly, a little fun if you're a math geek) to optimize your texture antialiasing to minimize it. In particular, the simplest yet most useful variation to explore is a "tweak" value that scales your texture spot size.

Team LRN

Even if you've been careful in determining spot size, it's very easy for the spot size to "drift" from its optimal value. It is not uncommon for the spot size to be off significantly! One renderer I deal with returns a spot size that is at least four times larger than it should be, and I have to scale it down appropriately to reduce the error.

With an error metric in hand, you can vary any part of your antialiasing code to determine whether it has a positive effect. Are there strange constants in your antialiasing code for dealing with spot size or sample rates? How about estimates of texture variability for use with index antialiasing? Cutoff frequencies for fractal noise? You can tweak all of these to optimize your algorithm's output.

You can also use the reference comparison to investigate the efficiency and quality of different algorithms. You can test a supersampling method to determine its speed and error as compared to a more intelligent but slower band-limiting method.

This method does not do well at catching *temporal* aliasing problems since it considers only a single image at a time. You could make a similar test that uses multiple images to determine the antialiasing error metric. Even this is not perfect, but it does help in better characterizing the final error.

EMERGENCY ALTERNATIVES

Sometimes with deadlines looming, you may still be tearing out your hair due to aliasing problems. While you can always throw more supersampling onto the problem, this often still isn't acceptable because of time or CPU limits. The ideas that follow may be technically dubious and even repugnantly crude, but elegance tends not to matter when you have a shot deadline in three hours.

The most common and obvious texture "tweak" is to scale the texture spot size. Even without a reference image comparison as discussed in the previous section, you can usually reduce aliasing artifacts by using an exaggerated spot size. Often this may eliminate aliasing at the expense of a softened look to your surface.

If your texture is aliasing and it uses a color map, you can try applying a blur to the color map. This will soften the transition zones between colors, which can hide a lot of terrible artifacts. Since the blur only has to be done once to the simple 1D color map, the computational overhead is inconsequential and does not slow final rendering. I first used this method to solve an aliasing problem, but I later found that it was a useful control for users to use at their own discretion.

A painful alternative that should only be used in true emergencies is a simple image blur effect. Make a mask that isolates the pixels that show the texture, and apply a 2D image blur to the final rendered image using that mask. This mask may be an

Team LRN

alpha channel that the renderer can output, or even a hand-painted one. By applying a small blur of just one or two pixels radius, aliasing artifacts in your texture are usually hidden very quickly. It has the unfortunate side effect of softening your surface features and even geometry details.

A final desperate alternative is effective with simple geometry. If you render your texture out as a 2D image, you can use the rendered image as a map on your surface. Most renderers have decent image map antialiasing using a MIP map or summed area table. Two-dimensional antialiasing with these methods tends to do especially well in high-compression areas, where a large amount of detail gets crammed into just a few pixels.

Team LRN

6

Team LRN

PRACTICAL METHODS FOR TEXTURE DESIGN

STEVEN WORLEY

INTRODUCTION

This entire book is primarily about texturing: different ways to use a mathematical formula to decide how to color the surface of an object. This chapter discusses aspects of texturing from a practical point of view, especially in dealing with the texture controls (parameters) users manipulate. This chapter shares many topics with Chapter 2, but from a different point of view.

I have written over 150 algorithmic surface textures (mostly for commercial use by other animators), and after a while you definitely get a feel for the design of the algorithms, as well as a toolbox full of utilities and ideas that make writing the textures easier. Several recurring themes and tricks occur over and over (such as mapping a computed value into a color lookup table or adding a bump-mapping effect to a color texture), and these topics form the basis of this chapter.

TOOLBOX FUNCTIONS

After building such a large library of textures, it has become clear that many new textures are just variants of each other—new ways to organize a set of stock routines together. Building blocks such as fractal noise functions, color mapping methods, and bump-mapping definitions occur in nearly every texture! This section discusses these common elements since their ubiquitous use makes them so important.

The Art of Noise

Fractal noise is, without question, the most important element currently used in procedural texturing. We won't discuss the basic implementation of the fractal noise function here, since many of the other chapters of this book discuss it in some detail (this alone is evidence of its importance in procedural texturing). Instead, we'll

Team LRN

discuss some enhancements and modifications to the basic noise algorithm, mostly to produce higher-quality and easier-to-use noise.

The biggest problem with the "plain" fractal noise algorithm is artifacts. The basic routine interpolates over a cubic lattice, and you'll be able to see that lattice on your surface, especially if you are using a small number of summed scales. Purists will also note that the basic Perlin noise isn't very isotropic, since diagonal directions have a longer distance gap between sample points than the points along the coordinate directions.

One method to hide this artifacting is to rotate each summed scale to align to a (precomputed) random orientation. Since the lattices of the different scales won't line up, the artifacts will at least be uncorrelated and a lot less noticeable. If you want to return derivatives for each scale (for bump mapping), you'll have to multiply the vector derivative result from each scale with the appropriate inverse rotation matrix (which is the transpose of the original rotation matrix) before you sum them.

I made these transformations by generating random rotation matrices and taking only those that point inside a single octant of a sphere. Since a 90-degree rotation (or 180 or 270) will still cause the lattices to match up, a single octant is the most rotation that is necessary.[1] Keeping the rotations confined within one octant also helps you check that there are not two similar rotations for different scales. (Remember this is a one-time precompute: you can manually sort through about 10 of these to make sure the matrices are all reasonably different.) To test what octant a rotation matrix maps to, just pump the vector (1 0 0) through the matrix and look for a vector with all positive components. You can generate the rotation matrices using the algorithm(s) in *Graphics Gems III* by Ken Shoemake and Jim Arvo. Figure 6.1 shows two examples of matrices that might be used.

This rotation trick is most useful when bump mapping, since the differentiation of the noise value clearly exposes the lattice artifacts: there are sharp discontinuities of the second derivatives along the lattice lines. When you differentiate for bump mapping, these second derivatives become discontinuous *first* derivatives and are easily visible.

Another strategy to help hide noise artifacts is to choose the lacunarity (the ratio between the sizes of successive scales) intelligently. A natural lacunarity to use is 0.5, *but don't use this!* It's better to use a value with a lot of digits (like 0.485743 or 0.527473), which will give you the same effective ratio. A ratio of exactly 0.5 will make the different scales "register" closely (the next smaller scale repeats exactly twice on top of the larger scale), so artifacts can appear periodically. This periodicity is broken by using a number that's not a simple ratio.

1. Note that these are full 3D rotations, not 2D, but you know what I mean.

Team LRN

Rotation matrix			Inverse		
0.98860	−0.07651	−0.12967	0.98860	0.07958	0.12780
0.07958	0.99665	0.01871	−0.07651	0.99665	−0.02881
0.12780	−0.02881	0.99138	−0.12967	0.01871	0.99138
0.85450	0.46227	−0.23691	0.85450	0.48691	0.18093
0.48691	−0.55396	0.67530	0.46227	−0.55396	−0.69240
0.18093	−0.69240	−0.69845	−0.23691	0.67530	−0.69845

FIGURE 6.1 Two random rotation matrices (and inverses).

Don't overlook enhancements or variants on the fractal noise algorithm. Extending the noise interpolation to four-dimensional space is particularly useful, as this allows you to make time-animated fractal noise. Another interesting variation I've used is to replace the random gradient and values of each noise "cell" with a lookup table of a function such as a sine wave. This lets you use your fractal noise machinery (and all the textures that have been designed to use it) with different patterns. Remember Perlin's original noise algorithm is not sacred; its usefulness should actually encourage you to experiment with its definition.

Color Mappings

Looking in our toolbox of common routines, the continual use of mappings makes this topic very important to discuss. Most textures use a paradigm that computes a value such as fractal noise and then uses this value to decide what color to apply to your object. In simpler textures this added color is always of a single shade, and the noise value is used to determine some "strength" from 0 to 1, which is used to determine how much to cross-fade the original surface color with the applied texture color.

This mapping allows the user quite a bit of control over the applied color, and its simplicity makes it both easy to implement and easy for the user to control even with raw numeric values as inputs. One mapping method I have used with great success defines four "transition" values. These transition values control the position and shape of a certain mapping function that turns the fractal noise (or other function) value into an output value from 0 to 1. Figure 6.2 shows the shape of this function. Two transitions, labeled T1 and T2, are each defined by a beginning value and ending value. By setting the different levels where these transitions occur, a large variety of mappings can be made from gradients to step functions to more complex "bandpass" shapes.

The implementation of such a mapping is trivial. The mapping is worth discussing mostly because this method is so useful in practice, since it is easy for users to

Team LRN

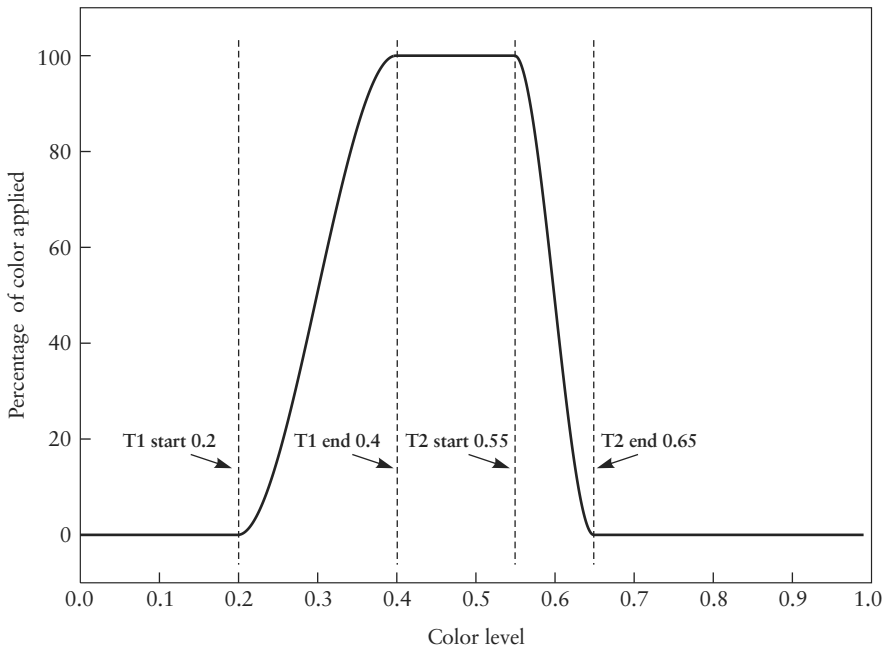


FIGURE 6.2 A mapping defined by four user values.

manipulate even numerically. A more sophisticated mapping method is much more general: color splines.

A color spline is simply an arbitrary mapping of an input value to an output color.[2] Often this mapping consists of a piecewise continuous interpolation between color values using a spline. Any number of spline knots (and node colors) can be defined, so the method is truly arbitrary. It does become very difficult for a user to control or manipulate an arbitrary spline mapping method without a good graphical interface; columns of color values and knot positions aren't easily visualizable.

2. You might recognize that the word "color" throughout this chapter (and book) is always used loosely. Most object surface attributes can be controlled through textures, and while the object's diffuse reflection color is the most often modified value, any of the other surface attributes are controllable using the exact same methods. It's just a lot easier to say "color" than "surface attributes specified by some vector that you can modify with your texture interface."

Team LRN

Bump-Mapping Methods

The most important element of *impressive* textures is the use of coordinated bump mapping. I'll repeat that: impressive and useful textures use bump mapping. Gouged-out depressions, ridges, or bumps are always more realistic and interesting than a simple layer of color paint. Luckily, nearly any color texture can have bump mapping added to it. Feedback from users shows that bump-mapping ability is unquestionably the most useful "extra" feature that a texture can have.

It's usually not difficult to add bump mapping to a basic texture. If you have a function that maps a scalar field to a value (like a color-mapped fractal noise value or a perturbed sine for marble veins), you can take the derivative of the scalar function in all three directions (x, y, z,), add these values to the surface normal, then renormalize it.[3] You should also add a parameter for users to control the apparent height of the bump. This is just multiplied to the derivative before the addition so the user can turn off the bump mapping by using a zero value or make the bump "go the other way" by using a negative value.

In the case of geometric figures (say, a hexagon mesh, or even a plain, checkerboard), there's not really a function to take the derivative of. In this case, I like making a "ridge," which is basically a line that follows along the exterior boundary of the figure at a fixed (user-defined) distance inside the boundary. This ridge basically makes a bevel around the outside rim of the figure, allowing the user to make the figure look raised or depressed into the surface. Figure 6.3 shows an example of how a simple bevel makes a flat pattern appear three-dimensional.

But a simple triangular bevel is awfully plain. It's good to give the user more control over the shape of this outer rim to make a variety of shapes, from a boxy groove to a circular caulk bead. In the case of a checkerboard, a user might make a concave rounded depression, which would make the squares look like they were surrounded by mortared joints.

I've tried several ideas to give the user control over the shapes of these ridges, but one has turned out to be the most useful. You don't want to have too many parameters for the user to juggle, but you still want to give them enough control to make a variety of joints. I've found that a combination of just four parameters works very well in defining a variety of bevel shapes.

The first parameter is the ridge width. What is the total width of the seam or bevel? In many (most!) cases this bevel is going to be butted up against another seam,

3. This is technically wrong! Really you need to do a little more work to get "correct" results, as discussed on page 170.

Team LRN

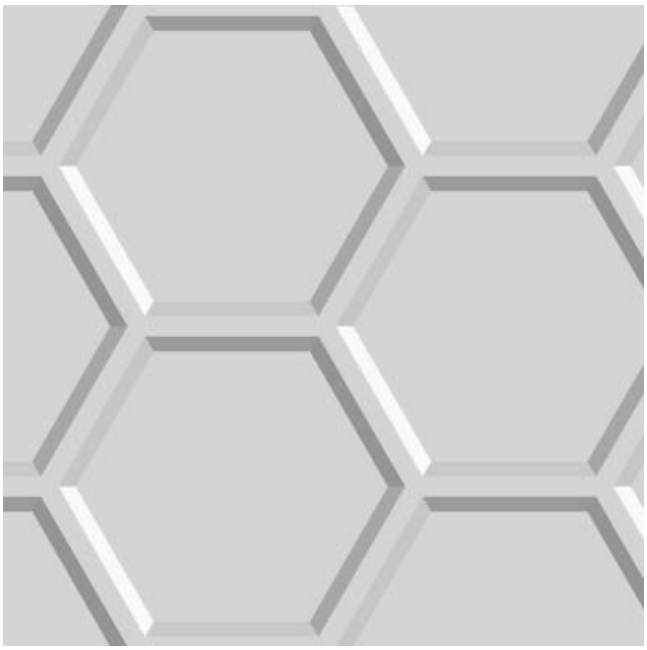


FIGURE 6.3 A ridged hexagon mesh.

so you might want to halve the number internally. (An example of this is a checkerboard: every tile is next to its neighbor. It's easier for the user to think about the total width of the joint instead of the width of the part of the joint in just one square.)

The second parameter is what I call the "plateau width." This is a distance along the outside of the ridge that isn't affected. This allows users to make joints that have a flat part in the middle of a seam, sort of a valley between the cliff walls that are formed by the bevel. This plateau obviously must be less than the total width of the bevel.

The last two parameters control the *shape* of the bevel. If you think of the ridge as being a fancy step function (it starts low, ends high, and does something in between), you want to be able to define that transition by a straight line, a smooth S curve, or anything in between. I've found that using a smooth cubic curve over the transition allows the users to define most useful shapes by setting just two numbers, the slopes of the curve at the start and end of the transition. Slopes of 0.0 would make a smooth blending with the rest of the surface, and if both slopes were 1.0, the

Team LRN

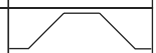
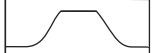
Bottom slope	Top slope	Plateau?	Ridge shape
1.0	1.0	Yes	
0.0	0.0	Yes	
0.0	1.0	Yes	
−1.0	−1.0	Yes	
0.0	0.0	No	

FIGURE 6.4 Bump-mapped ridge shapes for different slope controls.

transition would be a classic straight-line bevel. This is best shown in Figure 6.4, which also shows how the plateau is used.

The ridges shown are just examples, since there's really a continuum of curves that can be made. The slopes can even be continuously changed over time to animate the bevel morphing its shape.

These slope controls are obviously useful for making different ridge profiles, but how do you actually convert the parameters into numbers that can be used for bump mapping? The best way is to use the simple Hermite blending curves. Hermite curves are a simple type of spline, defined by a cubic polynomial over a range from 0 to 1. A cubic polynomial is very convenient since it's cheap to compute and has four degrees of freedom to control its shape. We can completely define this cubic curve by setting the starting and ending values and slopes.

This is perfect for us, since our ridge is basically a smooth curve that starts at a low height (0) and ends up high (1). We have the user specify the starting and ending slopes, defining the curve's last two degrees of freedom. The slope of this curve controls the amount of bump added to the surface normal.[4] We'll parameterize the width of the bevel from 0 to 1 to make using the Hermite spline easy.

We have four values (start and end heights, start and end slopes) we can use to construct the cubic polynomial by adding together weighted copies of the four

4. "Adding" is a tricky word! A crude literal addition of the derivative to the normal won't give you correct results (although they'll look OK sometimes). Again, see the bump-mapping coordinate discussion on page 170 to understand how bump mapping should be done correctly.

Team LRN

Hermite "blending functions." The sum, $F(t)$, is the unique cubic polynomial that satisfies the four constraints.

These blending functions are called P_1, P_4, R_1, and R_4, and represent the values of $F(0)$, $F(1)$, $F'(0)$, and $F'(1)$, respectively (see Figure 6.5).

$$
\begin{aligned}
P_1 &= 2t^3 - 3t^2 + 1 \\
P_4 &= -2t^3 + 3t^2 \\
R_1 &= t^3 - 2t^2 + t \\
R_4 &= t^3 - t^2 \\
P_1' &= 6t^2 - 6t \\
P_4' &= -6t^2 + 6t \\
R_1' &= 3t^2 - 4t + 1 \\
R_4' &= 3t^2 - 2t
\end{aligned}
$$

In the case of the ridge, the start and end values will always be fixed to 0 and 1, respectively. The user supplies the two slopes (call them s_b and s_t for bottom and top). The curve that represents our ridge shape is therefore $P_1 + s_b R_1 + s_t R_4$. The derivative of the curve at the hit point is $P_1'(t) + s_b R_1'(t) + s_t R_4'(t)$. This tells us the slope of our ridge at the hit point: the weight we use when adding to the surface normal.

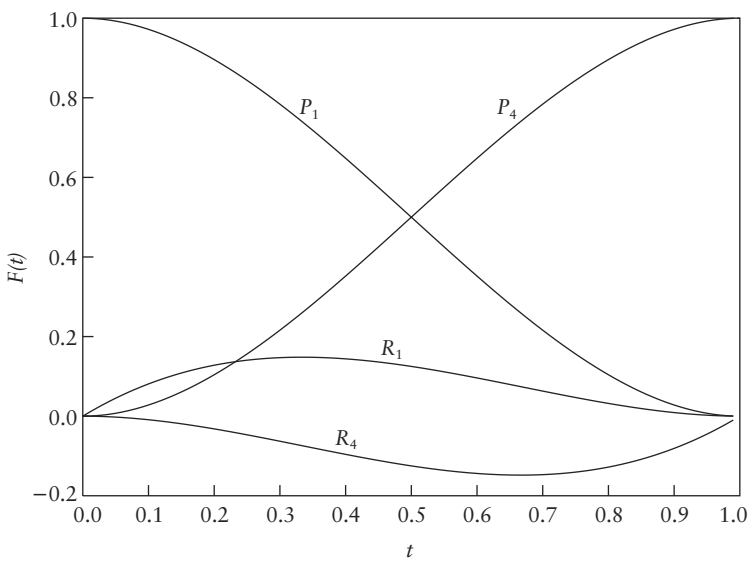


FIGURE 6.5 Hermite blending functions.

Team LRN

So, explicitly, the "ridge" algorithm is as follows:

1. Decide if the hit point lies within the bevel width. You'll need to come up with a measure of how close a hit point is to the "edge" of the figure.
2. If the hit is outside the bevel width or inside the plateau width, exit. The surface is flat at this location.
3. Set $\mathbf{N}_{add}$ equal to the normal vector perpendicular to the bevel. (This is the vector that points from the hit point toward the closest edge.) For example, on a checkerboard, if the hit point is on a bevel on the right (+X) side of a square, the normal would be (1, 0, 0), and the bevel on the bottom (−Y) of the square would have a normal of (0, −1, 0).
4. Set t to the normalized (dimensionless) distance along the slope. (If the hit point was d units from the edge, $t = 1.0 - (d -$ plateau width) /(ridge width − plateau width.)
5. Compute s, the slope of the cubic curve at the hit point. This is $P_1'(t) + s_b R_1'(t) + s_t R_4'(t)$, where s_b and s_t are user parameters.
6. Weight $\mathbf{N}_{add}$ by sA and add it to the hit point's surface normal. A is yet another user parameter, controlling the amplitude of the bump effect.
7. Renormalize the surface normal.

This is actually pretty easy to implement. The hardest part is probably determining the distance to the edge of a figure like a hexagon.

Note that this kind of bump mapping doesn't have to be restricted to geometric figures! It's easy to add to 3D scalar field functions, too. This lets you use something like fractal noise to make weird squiggly canyons or raise marble veins out of the surface of a stone. In this case the t variable is usually set by a high and low value defined by the user. The derivative of the field provides the $\mathbf{N}_{add}$ vector.

THE USER INTERFACE

Writing an algorithm that makes an interesting surface is only half the battle! One of the overlooked aspects of texture design is defining the user-accessible parameters. Even when an environment has a fancy GUI for manipulating the texture, the user is usually faced with setting a whole bunch of parameters with vague labels. The usability of your textures can be dramatically increased by using some thought and planning during your implementation. This is especially true when designing the

Team LRN

interface that you or users will use to control the texture. This interface often is not a fancy previewer, but simply a set of parameter values or even a crude text file. (In 1992, text file parameter definition was common. It's scary, but in 2002, it's still common!) Even with such basic interfaces, careful parameter design can make the texture significantly easier to use.

Parameter Ranges

It might seem obvious, but the user should be provided with suggestions for acceptable ranges for each parameter. Some ranges are easy enough to define, like colors or a percentage level. But, since all interesting textures seem to use ad hoc code that even the programmer doesn't quite understand, there are always going to be mystery constants that the user has to play with to get different effects. It's awfully inelegant to have a parameter named "squiggliness" that only makes useful patterns when it is between 55 and 107.[5]

One way to help the user control vague parameters like this is to *remap* a parameter's range. If "squiggliness" is just a mystery constant (which happens to be interesting over that 55–107 range), it's silly to expect users to enter these weird values. If you find that a parameter like this has one particular range of usability, you can use a linear transformation to remap the range: the user would enter a number from 0 to 1 for "squiggliness," not a value in the nonintuitive internal range. The new range of values provide the exact same amount of control as before, but at least the user knows that values like 0.3 or 0.9 might be interesting to try. (They can also experiment and try − 0.2 or 1.1, of course, but they'll understand if the result isn't useful.) The remapped range then serves as an indirect guide to selecting useful parameter settings.

Remapping linear ranges like this is easy, especially when you're remapping from 0 to 1. Since the transformation is linear, it's just a multiply and an addition. If x is between 0 and 1, you can map to the values from L and H by the simple formula $L + x\,(H - L)$.

This does *not* mean that you should make all parameters map to a 0–1 range! RGB colors are often easiest to enter as three 0–255 values. The width of a check in a checkerboard texture should be measured in the coordinate system's units. Common sense should be your guide.

5. This may seem like a contrived example, but veteran texture authors know it's not. Mystery constants seem to be an integral, recurring aspect of texture design.

Team LRN

Color Table Equalization

Color splines were discussed on page 182. Fractal noise textures in particular work well with this mapping method. However, while it is very easy to say, "Just feed the fractal noise value into a lookup table," how is this table designed? Obviously, it depends on whether you're making a texture that will be manipulated with a fancy GUI or one that is a subroutine the user has specified with raw parameter values. In practice, it usually boils down to the latter, since the GUI is just a front end for setting numeric parameters.

The biggest problem users find with setting color splines or levels is the fact that it is hard to know how much effect their color choices will have. This is critically true for renderers that can't quickly preview textures interactively. The biggest obstacle to users in setting color levels is that they do not know how much impact their choices will have on the rendered surface. Say that the users can define a color scheme where fractal noise below the value of -0.5 is black, a value above 0.5 is white, and in between values are a smooth gray transition. The problem is that the users have no idea how much of their object is going to be colored! Is the surface going to be dominated by huge patches of black and white, with narrow gray transitions in between? Or will it be gray values everywhere that never quite get to solid black or white?

Of course, the user can experiment and try to find the right levels to start and end color applications, but it is frustrating to have such an arbitrary, nonlinear scale! It is even worse, since if you change the number of noise octaves added onto the fractal sum, the scale changes again! If you think about it, it would be much easier if the users could deal with constant *percentage* levels and not weird, unknown ranges. Thus, a user could set the "lower" 20% of the noise to map to black, the "upper" 20% to white, and the remaining 60% to a gray gradient. This way the user has some idea how large each color range will be on the surface.

Thus there are two steps to making the noise levels "behave." First, they need to be normalized so that the amplitude ratio between scales and the total number of scales don't make the noise ranges change too much. The second part consists of making a table to map these values to percentages. The second step is really just a histogram equalization or a cumulative probability function.

The normalization step is pretty easy. Ideally, you want the fractal noise, no matter how many scales or what scale size or ratio you use, to return a value over the same range. You can do this adequately well by dividing the final sum by the square root of the sum of the squares of all the amplitudes of the scales. For those who don't want to parse that last sentence,

Team LRN

$$F(x) = \frac{\sum_{i=0}^{n-1} a^i N(\frac{x}{Ls^i})}{\sqrt{\sum_{i=0}^{n-1} (a^i)^2}}$$

where

F = normalized fractal noise value
L = largest scale size
s = size ratio between successive scales (usually around 0.5)
a = amplitude ratio between successive scales (usually around 0.5)
n = number of scales added (usually between 3 and 10)
x = hit location (a vector)

This normalization makes it a lot nicer when you are tweaking textures: the relative amounts of colors or size of the bumps won't be affected when you change the scale or amplitude ratio or number of summed scales.

The equalization step is a little more involved. The values of fractal noise will fall into a range, probably within −2 to 2, with most values around 0. It looks a lot like a normal distribution except the tails die out quickly. The histogram shown in Figure 6.6 shows the distribution of 10,000 sample values of fractal noise. In practice, more samples should be used, but with this number the slight random errors are visible to remind you this is an empirical measurement, not an analytic one.[6]

The histogram is very interesting, but we're really looking for a way to map the user's percentage level to a noise value. If, for example, 10% of all noise values fell below −0.9, we'd know to map 10% to that −0.9 value. Zero percent would map to the very lowest noise value, and 100% would map to the very highest. This is a *cumulative* function and is used in probability and statistics quite often. Luckily, the way to explicitly measure this relation empirically is straightforward.

First, store many random values of fractal noise in an array. Computing 250,000 samples is probably sufficient, but more is even better. (This is a one-time computation, so speed doesn't matter.) Sort the values into an ascending array. If you plotted the sorted array, you'd get a curve similar to the normalization plot shown in Figure 6.7. This is now a lookup table! For example, if you used 100,000 samples, the 50,000th number in the sorted array would correspond to the 50% level: half of the

6. You may recognize that in many cases the distribution of fractal functions tends to form a normal distribution because of the statistical Law of Large Numbers. So for some functions, you can just use a Gaussian as a distribution model and use the Erf() function to compute the normalization values.

Team LRN

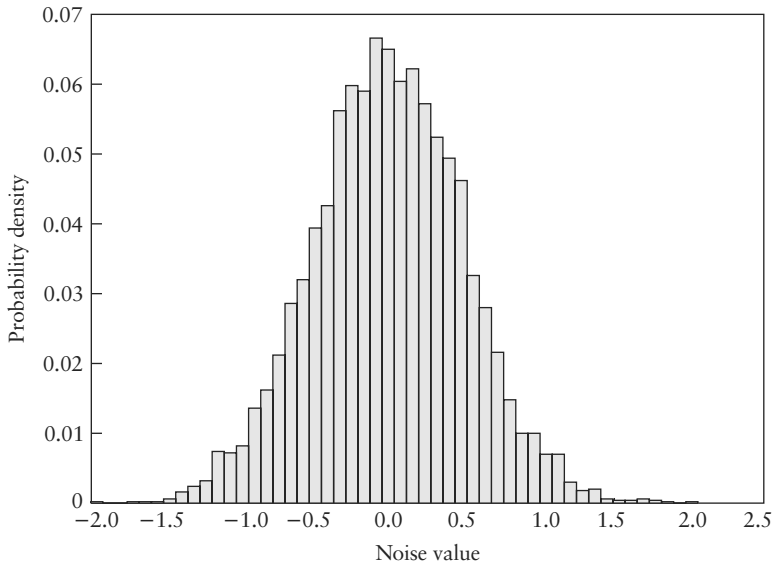


FIGURE 6.6 Histogram of 10,000 noise values.

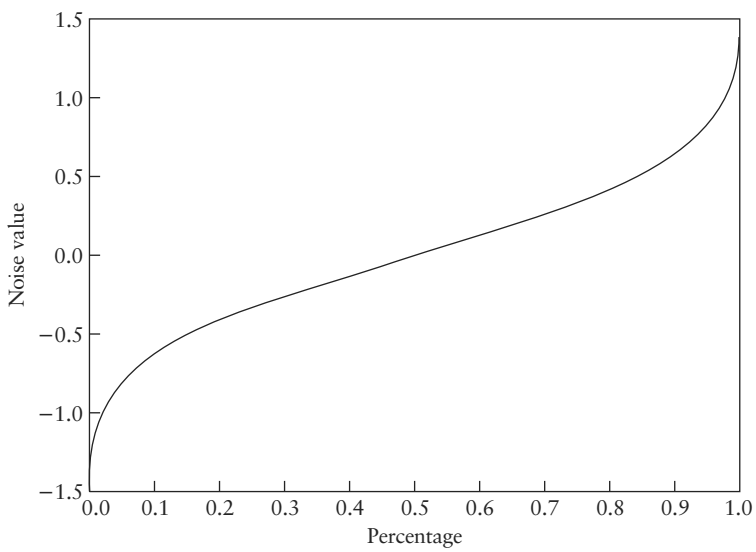


FIGURE 6.7 Percentage to noise value normalization curve.

Team LRN

noise values would be less than the value stored in this location in the sorted array. (It's the median!) The 10,000th number corresponds to the 10% level, since 10% of the 100,000 values are less than this value.

You obviously don't need to store 100,000 lookup values in your texture code! You can store perhaps just 11 equally spaced samples and linearly interpolate between them: a 15% level would be computed by averaging the 10% and 20% levels. Probably 16,000 values is a fine enough "grain" that you don't even need to bother interpolating.

Adding the translation from user parameter values (probably expressed in a 0 to 1 fractional value) to a noise value is easy enough, probably done as a first step and stored in a static variable. If you have a fancy GUI it can even be done as a preprocess as soon as the user has set the levels.

It is impossible to overstress the usefulness of this equalization to users. They can actually plan out the coverage of different colors fairly accurately the first time. If they want a sky 20% covered with clouds, they immediately know the settings to use. This convenience is no real computational load, but for the user it makes using the texture a *lot* more intuitive.

Exploring the Parameter Domain

Even as the author of a texture algorithm, determining appropriate parameter values to use can be very difficult. A mere texture *user* should have some strategies for manipulating a set of parameters.

The easiest method of setting a texture's parameters is to simply use previously interesting values! It may seem silly, but if you don't keep track of settings you have already produced, you'll always be starting from scratch whenever you wish to produce a new surface appearance. Keeping track of previous successful settings will give you a starting point for surfaces with the absolute minimum amount of future effort.

The library of settings is obviously not practical until you *build* that library first, and even then the library won't always have an example of the surface you desire. Other strategies for modifying the parameters give you significantly more control.

The most obvious but most demanding method is simply to understand the texture well and use your intelligence to choose the proper settings. When you are familiar with what each parameter does, you can predict what the modified surface will look like and just set the values appropriately. A trivial example is when you wish to change the colors applied by the texture, so you simply edit the color

Team LRN

parameters. However, knowledge of the texture's controls still isn't enough; if you want a surface that has a more "blobby" behavior, but there's no convenient parameter named "blobbiness," you'll be forced to determine what parameters (or set of parameters!) will produce the effect you want.

While in theory you can simply use your experience and judgment to set the parameters, in practice this can be difficult. Oddly, often the most effective editing method is just to vary different parameters wildly! Blindly adjust them! Don't be conservative! When the dust settles, look at the surface produced; it may be something you like and can use. If it's not, it only takes a few seconds to try again.

This random method is arguably just as powerful as the "forethought" editing method. Even when I feel I understand the texture well (because I wrote it!) random settings will sometimes make interesting appearances I didn't expect and couldn't even plan for. It's a great starting point, since from here you can use the "intellectual," planned editing steps. The random method will give you a wide variety of surfaces that, even if they are not immediately useful, you can still save in an attribute library; it's likely you might be able to use them sometime in the future.

The final editing strategy is what I call the "polish" step. When you have a setting you are happy with, you can use a methodical last pass to optimize your settings. The idea is that if the surface looks as good to you as possible, then changing any of the parameters must make a less perfect surface. (If it didn't, you'd have an even better surface than you started with, right?) With this in mind, you can go through each parameter sequentially. Perturb the value by a small amount, but enough to make a visible change. If you like the new pattern better, then congratulations, you've just refined your surface and improved it! If not, try adjusting the parameter a small amount the *other* way. If neither adjustment improves your surface, reset the value to what it was initially, then proceed to the next parameter. After you've gone through all the parameters, your surface is optimized; any parameter changes will make it less attractive.

While there are many strategies for editing parameters, the best strategy of all has got to be the long-term accumulation of a library of example parameter settings for each texture. You'll find new surfaces all the time when exploring (especially using the random flailing method), and if your attribute library is large, the next time you need a specific surface, you'll be that much more likely to have a good starting surface; otherwise you'd be forced to start from scratch.[7]

7. If you're designing a texture *system*, make sure to allow some sort of a user preset functionality!

Team LRN

Previews

Especially when you are initially developing textures, it is painful to run test after test to see how the texture is behaving. Obviously, the faster your development platform is, the better, but even then the design cycle time can be painfully slow.

I heartily agree with Ken Perlin's advice on this problem: generate *low*-resolution tests! Most problems are glaring errors, and even 200 × 200 pixel test images are often enough detail to judge the gross behavior of a texture very quickly. If you're in a compile-render-recode-compile loop, you might be surprised how much time is wasted just waiting for the next render, even with modern fast CPUs.

If you can't speed up your computer, the best way to help cut the design cycle time is to have the renderer display the image as it is rendering (if it can!). This way you can abort the render as soon as you know there is a problem, as opposed to waiting for the whole thing to finish before discovering there's a problem.

Even better is to design a previewer specifically for editing texture parameters. Place each parameter on a slider, and implement a progressive refinement display. This is a rendered image (perhaps just a simple plane or image of a sphere) that is rendered at a very coarse resolution. After the image has been computed, the image should be recomputed at a resolution that is twice as fine, updating the display appropriately. (Note that one-fourth of the pixels have already been computed!) This refinement continues until you've computed the texture down to the individual pixel level.

If this preview can be interrupted and restarted by changing a parameter (ideally by just grabbing and changing a slider), the design cycle for setting parameters becomes enormously faster. Even with a very slow computer, the update is interactive (due to the progressive display scheme). An editor like this increased the utility of my entire set of textures by a (conservatively estimated!) factor of five. *The utility of a texture to a user (especially a nontechnical one) is inversely proportional to the time for a complete design cycle.*

EFFICIENCY

As important and useful as it is, procedural texturing has one flaw that often limits its potential. That flaw is simply efficiency. A complex surface tends to require

Team LRN

complex computations to define it.[8] In particular, the most used building block of texturing (Perlin's fractal noise) is, unfortunately, slow.

The first and second editions of this book contained several pages of "speedup tricks" here, dealing with clever C macros to save a few microseconds. This was important in the "good old days" of 25 MHz processors. In today's modern era, we still love speed, but it's actually counterproductive to spend enormous effort optimizing C code when CPUs can now render a full screen of fractal noise in real time. This speed allows us to concentrate more on texture design and not programming details.

But efficiency is still important! It just means that we can focus more on the algorithms, especially caching and antialiasing. Because procedural textures are easier and faster to compute, we're just using them more. In 1989, Perlin's hypertextures were hyperslow. But today, most professional renderers have volumetric texture and shading effects, which are an order of magnitude more compute intensive.

TRICKS, PERVERSIONS, AND OTHER FUN TEXTURE ABUSES

Basic textures are often implemented in the same rough method. A function is passed an *XYZ* location in space, the current surface attributes, and perhaps some user parameters, and the texture modifies the attributes based on some internal function. This generic design is sufficient for more general surface texturing, but if you are creative you can actually produce some truly unique and interesting effects even with the limited information most textures usually have.

Volume Rendering with Surface Textures

One of the sneakiest tricks is to change a surface texture into a true volume texture. This has been done since the beginning of algorithmic texturing; Geoffrey Gardner, as early as 1985, was fabulously effective at turning simple ellipsoid surfaces into apparent cloud volumes (Gardner 1985). This wasn't a true volume integration, but it looked great!

You can actually do a true volume rendering with simple surface textures as long as you limit your surfaces to a somewhat restricted geometry. The easiest method of doing this relies on the texture knowing which direction the incoming ray is arriving from and assuming the user is applying the texture only to one type of surface like a sphere. The trick is simple: since you know the hit point of the ray, the direction of

8. The notable exceptions to this rule are patterns such as the fractal Mandelbrot set, but these have an uncontrollable complexity; you can't easily use the Mandelbrot set to make an image of granite paving stones.

Team LRN

the ray, and you're assuming you're hitting a sphere (so you know the geometry), you know exactly what path the ray would take if it passed through the sphere.

Now, if you perform a volume rendering with this information (which might be an analytic integration of a fog function, a numeric integration of a 3D volume density, or some other volume-rendering technique[9]), you can just use the final output color as a *surface* color. In particular, this is the surface *emittance;* often renderers will let you define this emittance (which is sometimes called *luminosity*) or, just as useful in this case, allow you to specify the true RGB surface color the surface will be treated as (and no shading will be done). By setting this surface color to the volume density computed intensity, the illusion of a gas can be complete! The gas must be contained by the sphere that defines it, but it is a true 3D effect and as such can be viewed from any angle. The biggest caveat is that the camera cannot fly through the volume density.

Odd Texture Ideas

If you think about the abilities of textures, you might realize that textures might be useful for more than just applying a color onto the surface of an object. In particular, if your host renderer passes the current surface color to the texture, you can design routines that can manipulate that color. You might map a bitmap onto a surface, then use a "gamma correction" texture to brighten the image. This texture would just be called after the brushmap is applied and would correct the image "on demand."

In fact, I've found quite a few "utility" textures like this that don't really apply a pattern at all. One is a solid color texture that has just two arguments, an RGB surface color and a "fade" value. By applying a solid color to an object, you can *cover up* previous textures. The fade value allows you to make the surface any opacity you want. If you linearly change this opacity over time, you can "fade in" an image map or previous texture. Or you could use the texture at a small (10%) opacity to just tint a surface's color.

Another useful texture transforms the RGB surface color to HSV, then lets the user "rotate" the hue around the color wheel. This is a cheap form of color cycling and is especially useful for animations.

You also shouldn't get stuck thinking that textures have to be fractal noise patterns or endlessly tiled figures. I've found that textures can be useful in adding fairly specific, structured features that you might think would be better implemented with an image map. One example is an LED display like a watch or calculator. The

9. Chapter 8 by David Ebert discusses this topic extensively.

Team LRN

texture takes a decimal number as an argument and displays it on the simulated seven-segment digits! This might seem weird, but you can now make something like an animated countdown without having to make hundreds of image maps manually.[10]

In a similar vein, a "radar" texture can make a sweeping line rotate around a circle tirelessly, with radar "blips" that brighten and fade realistically. All sorts of items that need to be continually updated (clocks, blinking control panel lights, simulated computer displays) can often be implemented as a semispecific surface texture.

A fun texture is the Mandelbrot set. Just map the hit point's *XY* position to the complex plane. As the camera approaches the surface of the object, the texture automatically zooms the detail of the set! This is an awfully fun way to waste valuable CPU time.

Don't get stuck in always layering an image onto the surface, either. One of my most often used textures uses fractal noise to *perturb* the brightness of the surface. A user might use a brushmap or another texture to provide surface detail, then use this "weathering" texture to add random variations to the surface. An example might be a texture that takes a fractal noise value F and scales the object surface color by a factor of $1 + \alpha F$. Even for subtle values of α the perfect hue of the surface is given some variation. This is dramatically impressive when a regular grid of squares suddenly becomes the weathered hull of an ocean liner.

2D Mapping Methods

For obvious reasons, 3D procedural textures are more versatile than 2D image maps. But modern renderers have powerful image mapping controls, which can often be great to use even with procedurals. The simple idea is to convert the 3D procedural texture into a 2D image map and allow the renderer to deal with it from there. This can speed up rendering and allow use of a procedural in 3D hardware (for OpenGL display), for video games that use real-time graphics, for solving antialiasing problems (since the renderer's image map antialiasing will be used), and of course for renderers that simply lack the proper texturing architecture.

Converting a 3D procedural to a 2D image requires some kind of harness to generate the maps. Tools for this are becoming more common as *UV* mapped modeling has become the norm. This means that the model itself has encoded UV values over each patch or polygon, correlating each surface point with a 2D image location. The image generation harness can iterate over the entire object, evaluating the

10. There is a sample of my LED texture on this book's Web site. It's admittedly a decade-old artifact of amateur coding, but may be interesting for reference.

Team LRN

procedural over the surface and "writing" the color values into the 2D image map. This test harness can be crude or sophisticated, depending on how it deals with sampling the surface evenly and how it filters the result to form the 2D texture sample. But once working, this process is invaluable for many artists' goals. The generic term for this technique is *texture baking*.

The 3D nature of the procedural texture allows many 2D effects that couldn't be made by hand-painting because of the varying distortion that a human artist can't compensate for. This is even true for common spherical projection maps that are just a special case of *UV* mapping (which use a formula instead of an arbitrary table to associate *UV* values with the surface.) For example, an unwrapped image of the Earth (showing continents, oceans, etc.) applied to a sphere produces a final spherical planet with the proper appearance. This polar coordinate mapping method allows the renderer to use the surface's 3D *XYZ* point to find the 2D *UV* coordinates of the corresponding position on the image map. This mapping is an "equal-angular" mapping, not a Mercator mapping, so the transformation is particularly simple:

$$U = \frac{\tan^{-1}(X, Y)}{2\pi}$$

$$V = \frac{\sin^{-1}(Z/\sqrt{X^2 + Y^2 + Z^2})}{2\pi} + 0.5$$

For *UV* maps defined by this spherical transform, we can invert the equations and use it to convert a *UV* map into an *XYZ* position for our procedural. Assuming we are evaluating the texture on the surface of a sphere of unit radius, this reverse transformation is not difficult. Simply,

$$X = \cos((V - 0.5)\cdot 2\pi)\sin(U\cdot 2\pi)$$

$$Y = \cos((V - 0.5)\cdot 2\pi)\cos(U\cdot 2\pi)$$

$$Z = \sin((V - 0.5)\cdot 2\pi)$$

Figures 6.8 and 6.9 show an example of how baking can work in practice. A basketball was defined algorithmically by using a brute-force procedure to evaluate a function over the surface of a sphere. Simple rules were used to determine whether a point was in a stripe or not (this is a quick geometric decision). A large lookup table of points distributed over the surface of the sphere was examined to see if the sample point was within a "pip" on the surface.[11] The inverse spherical mapping transformation was used to loop over the pixels of a texture map, and depending on whether the corresponding 3D surface point is located within a line or a pip, the

11. No subtlety here, every element in the lookup table of a couple thousand points was examined. This example was not a procedural texture designed for rendering (it was to produce a one-shot image map), so efficiency was not a big concern.

Team LRN

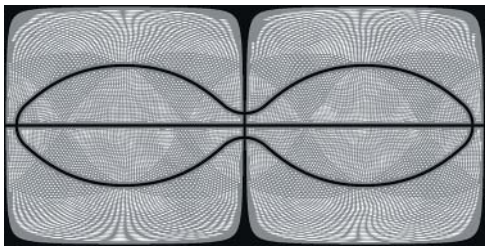


FIGURE 6.8 A 2D basketball image map.

FIGURE 6.9 The seamless wrapped ball.

image map was set to a gray value. This texture map was then used directly in a renderer.

In practice this trick works extremely well, since the predistortion of the image map exactly compensates for the renderer's subsequent distortion when wrapping. No seams are visible, and the image is not stretched; both occur when naive maps (such as digitized imagery) are used. This book's Web site contains the (admittedly old and inelegant) procedural to generate the basketball's appearance.

Note that this exact same method can be used to evaluate a texture over a cylinder or other shapes, as long as the transformation the renderer uses to map a 3D *XYZ* location to a *UV* image sample is known and an inverse transformation can be determined.

WHERE WE'RE GOING

Procedural texturing (and modeling) is certainly still in the frontier of computer graphics. It's an open-ended niche that is two decades old, but still growing in importance. The always-increasing power of computers is displacing rendering speed as the major bottleneck; the production of complex material appearances and

geometry is now possible and practical. Imagine the complexity present in a full model of a city; a single animator could not design each building and surface manually. The answer is of course procedural definition, which is why it is steadily becoming more and more important over time.[12]

But what topics will be at that frontier? There are obviously many, but I see several "holes" in classical texturing that will likely be important soon.[13] One of the largest gaps in even simple texturing is a strategy for more intelligent textures—ones that examine their environment (and particularly the object geometry) to change their appearance. The strategy used by Greg Turk (1991) for reaction-diffusion textures is one of the early forays into this field, but the potential is enormous. Imagine designing a dragon, but having the dragon scales behave differently over the belly of the dragon than the wings. Individual scales might change size based on the curvature of the skin (where there is high curvature, the plates need to be smaller to keep the area flexible). What is a good method for doing even this size scaling? What other effects could be modulated by object geometry? What about the geometry should the textures examine? The applications are obvious, but there have been no great developments in this area. Yet.

I see another topic becoming important soon. With the complexity of textures continually increasing (both in application methods and in internal design complexity), a "black-box" texture starts to become unwieldy for users to control. The black box is a texture personified by many control parameters, almost always numbers. When textures grow, the number of controls grows. It is not unusual to have up to 30 or 40 of these parameters! Even when carefully chosen and labeled, this many levers and knobs become difficult for the user to handle! This can be minimized through a good user interface (see page 194), but even then it is easy for a user (especially a nontechnical artist) to become swamped.

The black-box design is likely to be around for quite a while; it is a convenient way to design textures for the programmer. These parameters must therefore somehow be abstracted or hidden from users to avoid overwhelming them. This probably implies a user interface similar to the ones used by both Sims (1991a) and Todd and Latham (1993). Essentially, the user is presented not with many parameters to adjust

12. When this chapter was first written in 1994, I envisioned the day that we would have procedurally built cities, and even planets that could be viewed at any level of detail from orbit to 1 meter. That day has already come. At SIGGRAPH 2001, a procedural city technique was presented, as well as MojoWorld, a commercial "build a synthetic planet at any scale" renderer. What will happen in another 10 years?

13. I wrote this same sentence in 1994, but the topics are still just as appropriate in 2002, if not more so.

Team LRN

but with a collection of images. Each image is computed using different parameters (which are hidden from the user), and the user empirically ranks or sorts the images to identify the ones that are most interesting. From an artist's point of view, this is obviously appealing; it is very easy to use. The abstraction can also hide "parameters" of arbitrary complexity; Sims's textures are actually giant LISP expressions, not a mere vector of numeric parameters.

The big questions that remain are what sort of method is best for deciding how to choose the parameters to make the image to present to the user, and how to use the user's preferences to guide the production of new trial settings. Todd and Latham (1993) present a rudimentary answer to this question, mainly involving parameter "momentum," but especially for complex models this can become inadequate. I feel that development of a robust method for "texture evolution" based on numeric parameter vectors will be one of the most important tools for making procedural textures more useful from a user's point of view. In particular, two topics need to be studied: accounting for past explorations of the texture's parameter space and compensating for parameter correlations (where connections between parameters will complicate a "gradient" optimum search method). The growing complexity of textures really demands a new style of interface design, and this looks like the most promising.

So, while texturing is now two decades old, it is more important than ever. In computer graphics during the 1970s, the surface visibility problem was the main frontier. In the 1980s, lighting and surface properties (including the development of radiosity and BRDF surface models) were probably the most important developments. The 1990s produced efficient global lighting and image-based rendering. It is too early to tell what this current decade will be dominated by, but my prediction, especially for the next 10 years, will be procedural object and surface definition. We finally know how to render any model; now let's make the computer help us build the models themselves.

Team LRN

7

PROCEDURAL MODELING OF GASES

DAVID S. EBERT

INTRODUCTION

This chapter presents a framework for volumetric procedural modeling and texturing. Volumetric procedural models are a general class of procedural techniques that are great for modeling natural phenomena. Most graphics applications use surface-based models of objects; however, these models are not sufficient to effectively capture the intricate space-filling characteristics of many natural phenomena, such as water, fire, smoke, steam, clouds, and other gaseous phenomena. Volume models are also used extensively for modeling fur and other "soft" objects.

Volumetric procedural models use three-dimensional volume density functions ($vdf(x,y,z)$) that define the model density (or opacity) of a continuous three-dimensional space. Volume density functions (vdf's) are the natural extension of solid texturing (Perlin 1985) to describe the actual geometry of objects. I have used them for modeling and animating gases such as steam, fog, smoke, and clouds (Ebert, Carlson, and Parent 1994; Ebert 1991; Ebert and Parent 1990; Ebert, Ebert, and Boyer 1990; Ebert et al. 1997). Hypertextures (Perlin and Hoffert 1989), described by Ken Perlin in Chapter 12, and Inakage's flames (Inakage 1991) are other examples of the use of volume density functions.

This chapter will focus on the use of volumetric procedural models for creating realistic images of gases, specifically smoke, steam, and fog. I will use the term *gas* to encompass gas and particulate volumes, both of which are governed by light-scattering models for small particles. Atmospheric attenuation and lighting models are robust enough to encompass both types of volumes and produce visually correct results.

As in the preceding chapters, the procedures in this chapter will make use of the stochastic functions `noise( )` and `turbulence( )`. I will give a simple implementation of the `noise( )` and `turbulence( )` functions used in my system.

Team LRN

This chapter first summarizes previous approaches to modeling gases, and then presents a brief description of my volume ray-tracing system for gases and several approaches for using graphics hardware to implement these effects. The concept of three-dimensional "solid spaces" is introduced next to stress the importance of the relationship of procedural (function/texture) space to object and screen space. Finally, it concludes with a detailed description of how to create still images of gases.

PREVIOUS APPROACHES TO MODELING GASES

Attempts to model gases in computer graphics started in the late 1970s. Since that time, there have been many different approaches. These can be categorized as techniques for modeling the geometry of gases and techniques for rendering scenes with gases and atmospheric effects.

There have been several approaches to modeling the geometry of gases. Some authors use a constant density medium (Klassen 1987; Nishita, Miyawaki, and Nakamae 1987) but do allow different layers of constant densities. Still, only very limited geometries for gases can be modeled. Voss and Musgrave use fractals to create realistic clouds and fog effects (Voss 1983; Musgrave 1990). Max (1986) uses height fields for simulating light diffusion effects, and Kajiya uses a physically based model for modeling clouds (Kajiya and Von Herzen 1984). Gardner (1985, 1990) has produced extremely realistic images of clouds by using Fourier synthesis to control the transparency of hollow ellipsoids. The main disadvantage of this approach is that it is not a true three-dimensional geometric model for the clouds.

I have developed several approaches for modeling gases based on volume density functions (Ebert 1991; Ebert and Parent 1990; Ebert, Ebert, and Boyer 1990; Ebert, Boyer, and Roble 1989). These models are true three-dimensional models for the geometry of gases and provide more realistic results than previous techniques. Stam and Fiume (1991, 1993, 1995) also use a three-dimensional geometric model for gases. This model uses "fuzzy blobbies," which are similar to volumetric metaballs and particle systems, for the geometric model of the gases. Stam and Fedkiw have extended this work to use physically based Navier-Stokes solutions for modeling gases and have achieved very realistic effects (Stam 1999; Fedkiw, Stam, and Jensen 2001). Sakas (1993) uses spectral synthesis to define three-dimensional geometric models for gases. Many authors have used a variety of techniques for the detailed modeling and real-time approximation of clouds, which is described in Chapter 9.

Team LRN

The rendering of scenes containing clouds, fog, atmospheric dispersion effects, and other gaseous phenomena has also been an area of active research in computer graphics. Several papers describe atmospheric dispersion effects (Willis 1987; Nishita, Miyawaki, and Nakamae 1987; Rushmeier and Torrance 1987; Musgrave 1990), while others cover the illumination of these gaseous phenomena in detail (Blinn 1982a; Kajiya and Von Herzen 1984; Max 1986; Klassen 1987; Ebert and Parent 1990). Most authors use a low-albedo reflection model, while a few (Blinn 1982a; Kajiya and Von Herzen 1984; Rushmeier and Torrance 1987; Max 1994; Nishita, Nakamae, and Dobashi 1996; Wann Jensen and Christensen 1998; Fedkiw, Stam, and Wann Jensen 2001) discuss the implementation of a high-albedo model. (A low-albedo reflectance model assumes that secondary scattering effects are negligible, while a high-albedo illumination model calculates the secondary and higher-order scattering effects.) There has also been considerable work in the past several years in developing interactive rendering techniques for gases and clouds, described in Chapter 10.

THE RENDERING SYSTEM

For true three-dimensional images and animations of gases, volume rendering must be performed. Any volumetric rendering system, such as the systems described by Perlin and Hoffert (1989) and Kajiya and Von Herzen (1984), or approximated volume-rendering system can be used, provided that you can specify procedures to define the density/opacity of each volume element for the gas. I will briefly discuss my rendering approach, which is described in detail in Ebert and Parent (1990). This hybrid rendering system uses a fast scanline a-buffer rendering algorithm for the surface-defined objects in the scene, while volume-modeled objects are volume rendered using a per-pixel volume ray-tracing technique. The algorithm first creates the a-buffer for a scanline containing a list for each pixel of all the fragments that partially or fully cover the pixel. Then, if a volume is active for a pixel, the extent of volume rendering necessary is determined. The volume rendering is performed next, creating a-buffer fragments for the separate sections of the volumes. (It is necessary to break the volume objects into separate sections that lie in front of, in between, and behind the surface-based fragments in the scene to generate correct images.) The volume ray tracing that is used is a very simple extension to a traditional ray tracer: instead of stopping the tracing of a ray when an object is hit, the tracing actually steps through the object/volume at a defined step size and accumulates the opacity and color of each small volumetric segment of the object/volume. Volume rendering ceases once full coverage of the pixel by volume or surfaced-defined elements is

Team LRN

achieved. Finally, these volume a-buffer fragments are sorted into the a-buffer fragment list based on their average Z-depth values, and the a-buffer fragment list is rendered to produce the final color of the pixel.

Volume-Rendering Algorithm

The volume-rendering technique used for gases in this system is similar to the one discussed in Perlin and Hoffert (1989). The ray from the eye through the pixel is traced through the defining geometry of the volume. For each increment through the volume sections, the volume density function is evaluated. The color, density, opacity, shadowing, and illumination of each sample is then calculated. The illumination and densities are accumulated based on a low-albedo illumination model for gases and atmospheric attenuation.

The basic gas volume-rendering algorithm is the following:

```
for each section of gas
  for each increment along the ray
     get color, density, & opacity of this element
     if self_shadowing
        retrieve the shadowing of this element from the
            solid shadow table
     color = calculate the illumination of the
             gas using opacity, density, and
             the appropriate model
     final_clr = final_clr + color;
     sum_density = sum_density + density;
     if( transparency < 0.01)
        stop tracing
    increment sample_pt
  create the a_buffer fragment
```

In sampling along the ray, a Monte Carlo method is used to choose the sample point to reduce aliasing artifacts. The opacity is the density obtained from evaluating the volume density function multiplied by the step size. This multiplication is necessary because in the gaseous model we are approximating an integral to calculate the opacity along the ray (Kajiya and Von Herzen 1984). The approximation used is

$$\text{opacity} = 1 - e^{-\tau \times \sum_{t_{near}}^{t_{far}} \rho(x(t), y(t), z(t)) \times \Delta t}$$

where τ is the optical depth of the material, $\rho(\ )$ is the density of the material, t_{near} is the starting point for the volume tracing, and t_{far} is the ending point. The final increment along the ray may be smaller, so its opacity is scaled proportionally (Kajiya and Kay 1989).

Team LRN

Illumination of Gaseous Phenomena

The system uses a low-albedo gaseous illumination model based on Kajiya and Von Herzen (1984). The phase functions that are used are sums of Henyey-Greenstein functions as described in Blinn (1982a). The illumination model is the following:

$$B = \sum_{t_{near}}^{t_{far}} e^{-\tau \times \sum_{t_{near}}^{t} \rho(x(u)y(u),z(u)) \times \Delta u} \times I \times \rho(x(t), y(t), z(t)) \times \Delta t$$

where I is

$$\sum_i I_i(x(t), y(t), z(t)) \times phase(\theta)$$

$phase(\theta)$ is the phase function, the function characterizing the total brightness of a particle as a function of the angle between the light and the eye (Blinn 1982a). $I_i(x(t),y(t),z(t))$ is the amount of light from light source I reflected from this element.

Self-shadowing of the gas is incorporated into I by attenuating the brightness of each light. An approximation for a high-albedo illumination model can also be incorporated by adding an ambient term based on the albedo of the material into I_i. This ambient term accounts for the percentage of light reflected from the element due to second- and higher-order scattering effects.

Volumetric Shadowing

Volumetric shadowing is important in obtaining accurate images. As mentioned above, self-shadowing can be incorporated into the illumination model by attenuating the brightness of each light. The simplest way to self-shadow the gas is to trace a ray from each of the volume elements to each of the lights, determining the opacity of the material along the ray using the preceding equation for opacity. This method is similar to shadowing calculations performed in ray tracing and can be very slow. My experiments have shown that ray-traced self-shadowing can account for as much as 75% to 95% of the total computation time.

To speed up shadowing calculations, a precalculated table can be used. Kajiya discusses this approach with the restriction that the light source be at infinity (Kajiya and Von Herzen 1984; Kajiya and Kay 1989). I have extended this approach to remove this restriction. Using my technique, the light source may even be inside the volume. This shadow-table-based technique can improve performance by a factor of 10–15 over the ray-traced shadowing technique. A complete description of this shadowing technique can be found in Ebert (1991).

Team LRN

The shadow table is computed once per frame. To use the shadow table when volume tracing, the location of the sample point within the shadow table is determined. This point will lie within a parallelepiped formed by eight table entries. These eight entries are trilinearly interpolated to obtain the sum of the densities between this sample point and the light. To determine the amount of light attenuation, the following formula is used.

$$\text{light\_atten} = 1 - e^{-\tau \times \text{sum\_densities} \times \text{step\_size}}$$

As mentioned above, this shadow table algorithm is much more efficient than the ray-tracing shadowing algorithm. Another benefit of this approach is the flexibility of detail on demand. If very accurate images are needed, the size of the shadow table can be increased. If the volume is very small in the image and very accurate shadows are not needed, a small resolution shadow table (e.g., 8^3) can be chosen. For most images, I use a shadow table size of 32^3 or 64^3.

Recent work in shadowing for volumetric objects provides more efficient shadow rendering. The deep shadow map approach is an extension of traditional two-dimensional texture mapping that allows shadows from semitransparent volumetric objects (Lokovic and Veach 2000) by storing a visibility function in each entry in the deep shadow map. This visibility function stores, for each depth, the fraction of light that reaches this depth.[1] Kim and Neumann (2001) have developed a hardware-accelerated opacity shadow mapping technique that is similar to 3D hardware texture-based volume rendering, using a large number (e.g., 100–500) of opacity maps to accurately calculate volumetric shadows. Kniss, Kindlmann, and Hansen (2002) have recently developed a 3D hardware texture map slicing technique that allows shadows from volumetric objects, including gases that are rendered using three-dimensional texture maps.

ALTERNATIVE RENDERING AND MODELING APPROACHES FOR GASES

There are three types of alternative rendering approaches commonly used for gases:

- Particle systems
- Billboards and imposters
- Three-dimensional hardware texture mapping

1. In actuality, a piecewise linear approximation of the function is stored.

Team LRN

Particle systems are most commonly used for thin gases, such as smoke. There are two problems in using particle systems for gases. The first is the complexity of computing particle self-shadowing, and the second is the computational complexity of simulating large or dense areas of gas (millions of particles may be needed for the simulation). Billboards and imposters have been effectively used for interactive cloud rendering, with some limitations imposed on the animation of the clouds and/or light sources for efficiency in interactive rendering (Dobashi et al. 2000; Harris and Lastra 2001). Three-dimensional hardware texture mapping can be used with slice-based volume rendering to simulate clouds and other dense gases (Kniss, Kindlmann, and Hansen 2002). A combination of a coarse volume representation and procedural detail, which can be rendered in the graphics processor, is used to produce convincing volumetric effects and is described in more detail in Chapter 10.

A PROCEDURAL FRAMEWORK: SOLID SPACES

This section describes a general and flexible framework for procedural techniques, called *solid spaces*. The development of this framework, its mathematical definition, and its role in procedural texturing and modeling are described below.

Development of Solid Spaces

My approach to modeling and animating gases started with work in solid texturing. Solid texturing can be viewed as creating a three-dimensional color space that surrounds the object. When the solid texture is applied to the object, it is as if the defining space is being carved away. A good example of this is using solid texturing to create objects made from wood and marble. The solid texture space defines a three-dimensional volume of wood or marble, in which the object lies.

Most of my solid texturing procedures are based on the noise and turbulence functions. This work extended to modeling gases when I was asked to produce an image of a butterfly emerging from fog or mist. Since gases are controlled by turbulent flow, it seemed natural to somehow incorporate the use of noise and turbulence functions into this modeling. My rendering system already supported solid texturing of multiple object characteristics, so the approach that I developed was to use solid textured transparency to produce layers of fog or clouds. The solid textured transparency function was, of course, based on turbulence. This approach is very similar to Gardner's approach (Gardner 1985) and has the same disadvantage of not being a true three-dimensional model, even though the solid texture procedure is defined throughout three-space. In both cases, these three-dimensional procedures are evaluated only at the surfaces of objects. To remedy this shortcoming, my next extension

Team LRN

was to use turbulence-based procedures to define the density of three-dimensional volumes, instead of controlling the transparency of hollow surfaces.

As you can see, the idea of using three-dimensional spaces to represent object attributes such as color, transparency, and even geometry is a common theme in this progression. My system for representing object attributes using this idea is termed *solid spaces*. The solid space framework encompasses traditional solid texturing, hypertextures, and volume density functions within a unified framework.

Description of Solid Spaces

Solid spaces are three-dimensional spaces associated with an object that allow for control of an attribute of the object. For instance, in solid color texturing, described in Chapters 2 and 6, the texture space is a solid space associated with the object that defines the color of each point in the volume that the object occupies. This space can be considered to be associated with, or represent, the space of the material from which the object is created.

Solid spaces have many uses in describing object attributes. As mentioned earlier, solid spaces can be used to represent the color attributes of an object. This is very natural for objects whose color is determined from procedures defining a marble color space, as in Figure 8.2. Many authors use solid color spaces for creating realistic images of natural objects (Perlin 1985; Peachey 1985; Musgrave and Mandelbrot 1989). Often in solid texturing (using solid color spaces) there are additional solid spaces, which are combined to define the color space. For example, in most of my work in solid texturing, a noise and turbulence space is used in defining the color space. Other solid space examples include geometry (hypertextures and volume density functions), roughness (solid bump mapping), reflectivity, transparency, illumination characteristics, and shadowing of objects. Solid spaces can even be used to control the animation of objects, as will be described in the next chapter.

Mathematical Description of Solid Spaces

Solid spaces can be described simply in mathematical terms. They can be considered to be a function from three-space to n-space, where n can be any nonzero positive integer. More formally, solid spaces can be defined as the following function:

$$S(x,y,z) = F,\ F \in R^n,\ n \in 1, 2, 3, \ldots$$

Of course, the definition of the solid space can change over time; thus, time could be considered to be a fourth dimension to the solid space function. For most uses of solid spaces, S is a continuous function throughout three-space. The

Team LRN

exception is the use of solid spaces for representing object geometries. In this case, *S* normally has a discontinuity at the boundary of the object. For example, in the case of implicit surfaces, *S* is normally continuous throughout the surface of the object, but thresholding is used to change the density value abruptly to 0 for points whose density is not within a narrow range of values that defines the surface of the object. The choice of *F* determines the frequencies in the resulting solid spaces and, therefore, the amount of aliasing artifacts that may appear in a final image.

GEOMETRY OF THE GASES

Now that some background material has been discussed, this section will describe detailed procedures for modeling gases. As mentioned in the introduction, the geometry of the gases is modeled using turbulent-flow-based volume density functions. The volume density functions take the location of the point in world space, find its corresponding location in the turbulence space (a three-dimensional space), and apply the turbulence function. The value returned by the turbulence function is used as the basis for the gas density and is then "shaped" to simulate the type of gas desired by using simple mathematical functions. In the discussion that follows, I will first describe my noise and turbulence functions and then describe the use of basic mathematical functions for shaping the gas. Finally, the development of several example procedures for modeling the geometry of the gases will be explored.

My Noise and Turbulence Functions

In earlier chapters of this book, detailed descriptions of noise and turbulence were discussed, including noise and turbulence functions with much better spectral characteristics. I am providing my implementations to enable the reader to reproduce the images of gases described in this chapter. If other noise implementations are used, then the gas shaping that is needed will be slightly different. (I have experimented with this.) My noise implementation uses trilinear interpolation of random numbers stored at the lattice points of a regular grid. I use a grid size of 64 × 64 × 64. The 3D array is actually 65 × 65 × 65 with the last column equaling the first column to make accessing entries easier (`noise[64][64][64] = noise[0][0][0]`). To implement this using 3D texture mapping hardware, you can simply create the 64 × 64 × 64 table and turn the texture repetition mode to repeat. This random number lattice-based noise implementation is actually very well suited for 3D texture mapping hardware implementation, and the simple DirectX or OpenGL calls to read values from this 3D texture map will perform the noise lattice interpolation automatically.

The noise lattice is computed and written to a file using the following code:

```
// ////////////////////////////////////////////////////
//                  WRITE_NOISE.C
// This program generates a noise function file for solid texturing.
//             by David S. Ebert
// ////////////////////////////////////////////////////
#include <math.h>
#include <stdio.h>
#define SIZE 64
double drand48();
int main(int argc, char **argv )
{
  long i,j, k, ii,jj,kk;
  float noise[SIZE+1][SIZE+1][SIZE+1];
  FILE *noise_file;
  noise_file = fopen("noise.data","w");

  for (i=0; i<SIZE; i++) for
    (j=0; j<SIZE; j++) for
    (k=0; k<SIZE; k++)
       {
        noise[i][j][k] = (float)drand48( );
       }
// This is a hack, but it works. Remember this is
// only done once.
 for (i=0; i<SIZE+1; i++)
   for (j=0; j<SIZE+1; j++)
     for (k=0; k<SIZE+1; k++)
      {
       ii = (i == SIZE)? 0: i;
       jj = (j == SIZE)? 0: j;
       kk = (k == SIZE)? 0: k;
       noise[i][j][k] = noise[ii][jj][kk];
      }
   fwrite(noise,sizeof(float),(SIZE+1)*(SIZE+1)*(SIZE+1),
        noise_file);
fclose(noise_file);
}
```

To compute the noise for a point in three-space, the `calc_noise( )` function given below is called. This function replicates the noise lattice to fill the positive octant of three-space. To use this procedure, the points must be in this octant of space. I allow the user to input scale and translation factors for each object to position the object in the noise space.

The noise procedure given below, `calc_noise`, uses trilinear interpolation of the lattice point values to calculate the noise for the point. The `turbulence( )` function given below is the standard Perlin turbulence function (Perlin 1985).

Team LRN

```
typedef struct xyz_td
 {
  float x, y, z;
 } xyz_td;
float calc_noise( );
float turbulence( );

// ////////////////////////////////////////////////////
//                   Calc_noise
// This is basically how the trilinear interpolation works. I
// lerp down the left front edge of the cube, then the right
// front edge of the cube(p_l, p_r). Then I lerp down the left
// back and right back edges of the cube (p_l2, p_r2). Then I
// lerp across the front face between p_l and p_r (p_face1). Then
// I lerp across the back face between p_l2 and p_r2 (p_face2).
// Now I lerp along the line between p_face1 and p_face2.
// ////////////////////////////////////////////////////
float calc_noise(xyz_td pnt)
{
  float t1;
  float p_l,p_l2,// value lerped down left side of face 1 & face 2
       p_r,p_r2, // value lerped down left side of face 1 & face 2
       p_face1,  // value lerped across face 1 (x-y plane ceil of z)
       p_face2,  // value lerped across face 2 (x-y plane floor of z)
       p_final;  //value lerped through cube (in z)
  extern float noise[SIZE+-1][SIZE+-1][SIZE+-1];
  register int x, y, z, px, py, pz;

  px = (int)pnt.x;
  py = (int)pnt.y;
  pz = (int)pnt.z;
  x = px &(SIZE); // make sure the values are in the table
  y = py &(SIZE); // Effectively replicates table throughout space
  z = pz &(SIZE);

  t1  =     pnt.y - py;
  p_l =  noise[x][y][z+1]+t1*(noise[x][y+1][z+1]-
         noise[x][y][z+1]);
  p_r =  noise [x+1][y][z+1]+t1*(noise[x+1][y+1][z+1]-
         noise[x+1][y][z+1]);
  p_l2 = noise[x][y][z]+ t1*(noise[x][y+1][z] -
         noise[x][y][z]);
  p_r2 = noise[x+l][y][z]+ t1*(noise[x+1][y+1][z]-noise[x+1][y][z]);
  t1 = pnt.x - px;
  p_face1 = p_l + t1 * (p_r - p_l);
  p_face2 = p_l2 + t1 * (p_r2 - p_l2);
  t1 = pnt.z - pz;
  p_final = p_face2 + t1*(p_face1 - p_face2);
  return(p_final);
}
```

```
//
// ////////////////////////////////////////////////
//             TURBULENCE
// ////////////////////////////////////////////////
float turbulence(xyz_td pnt, float pixel_size)
{
  float t, scale;
  t=0;
  for(scale=1.0; scale >pixel_size; scale/=2.0)
    {
     pnt.x = pnt.x/scale; pnt.y = pnt.y/scale;
     pnt.z = pnt.z/scale;
     t+= calc_noise(pnt)* scale;
    }
  return(t);
}
```

Neither of these routines is optimized. Using bit-shifting operations to index into the integer lattice can optimize the noise lattice access. Precalculating a table of scale multipliers and using multiplication by reciprocals instead of division can optimize the turbulence function.

Basic Gas Shaping

Several basic mathematical functions are used to shape the geometry of the gas. The first of these is the power function. Let's look at a simple procedure for modeling a gas and see the effects of the power function, and other functions, on the resulting shape of the gas.

```
void basic_gas(xyz_td pnt, float *density,float *parms)
{
 float turb;
 int i;
 static float pow_table[POW_TABLE_SIZE];
 static int calcd=1;

 if(calcd) { calcd=0;
   for(i=POW_TABLE_SIZE-1; i>=0; i--)
      pow_table[i]=(float)pow(((double)(i))/(POW_TABLE_SIZE-1)*
                  parms[1]*2.0,(double)parms[2]);
   }
 turb =turbulence(pnt, pixel_size);
 *density =pow_table[(int)(turb*(.5*(POW_TABLE_SIZE-1)))];
}
```

This procedure takes as input the location of the point being rendered in the solid space, `pnt`, and a parameter array of floating-point numbers, `parms`. The

returned value is the density of the gas. `parms[1]` is the maximum density value for the gas with a range of 0.0–1.0, and `parms[2]` is the exponent for the power function.

Figure 7.1 shows the effects of changing the power exponent, with `parms[1] = 0.57`. Clearly, the greater the exponent, the greater the contrast and definition to the gas plume shape. With the exponent at 1, there is a continuous variation in the density of the gas; with the exponent at 2, it appears to be separate individual plumes of gas. Therefore, depending on the type of gas being modeled, the appropriate exponential value can be chosen. This procedure also shows how precalculated tables can increase the efficiency of the procedures. The `pow_table[ ]` array is calculated once per image and assumes that the maximum density value, `parms[1]`, is constant for each given image. A table size of 10,000 should be sufficient for producing accurate images. This table is used to limit the number of `pow` function calls. If the following straightforward implementation were used, a power function call would be needed per volume density function evaluation:

```
*density = (float) pow((double)turb*parms[1],(double)parms[2]);
```

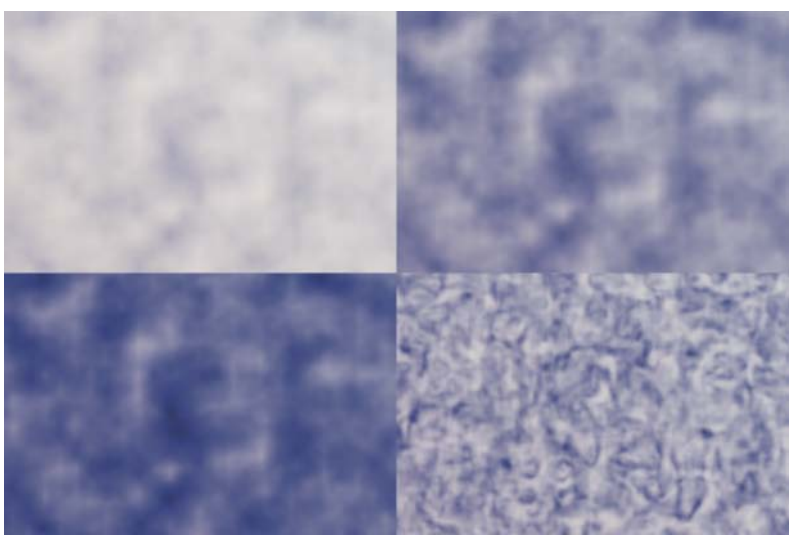


FIGURE 7.1 The effects of the power and sine function on the gas shape: (top left) a power exponent of 1; (top right) a power exponent of 2; (bottom left) a power exponent of 3; (bottom right) the sine function applied to the gas. Copyright © 1994 David S. Ebert.

Team LRN

Assuming an image size of 640 × 480, with 100 volume samples per pixel, the use of the precomputed table saves 30,710,000 `pow` function calls.

Another useful mathematical function is the sine function. Perlin (1985) uses the sine function in solid texturing to create marble, which will be described in a later section. This function can also be used in shaping gases, which can be accomplished by making the following change to the `basic_gas` function:

```
turb =(1.0 +sin(turbulence(pnt, pixel_size)*M_PI*5))*.5;
```

This change creates "veins" in the shape of the gas, similar to the marble veins in solid texturing. As can be seen from these examples, it is very easy to shape the gas using simple mathematical functions. The remainder of this chapter will extend this `basic_gas` procedure to produce more complex shapes in the gas.

Patchy Fog

The first example of still gas is patchy fog. The earlier `basic_gas` function can be used to produce still images of patchy fog. For nice fog, `parms[1]=0.5, parms[2]= 3.0`. The `parms[2]` value determines the "patchiness" of the fog, with lower values giving more continuous fog. `parms[1]` controls the denseness of the fog that is in the resulting image.

Steam Rising from a Teacup

The goal of our second example is to create a realistic image of steam rising from a teacup. The first step is to place a "slab" (Kajiya 1986) of volume gas over the teacup. (Any ray-traceable solid can be used for defining the extent of the volume.) As steam is not a very thick gas, a maximum density value of 0.57 will be used with an exponent of 6.0 for the power function. The resulting image in Figure 7.2 (left) was produced from the preceding `basic_gas` procedure.

The image created, however, does not look like steam rising from a teacup. First, the steam is not confined to only above and over the cup. Second, the steam's density does not decrease as it rises. These problems can be easily corrected. To solve the first problem, ramp off the density spherically from the center of the top of the tea. This will confine the steam within the radius of the cup and make the steam rise higher over the center of the cup. The following `steam_slab1` procedure incorporates these changes into the `basic_gas` procedure:

```
void steam_slab1(xyz_td pnt, xyz_td pnt_world, float
                 *density, float *parms, vol_td vol)
```

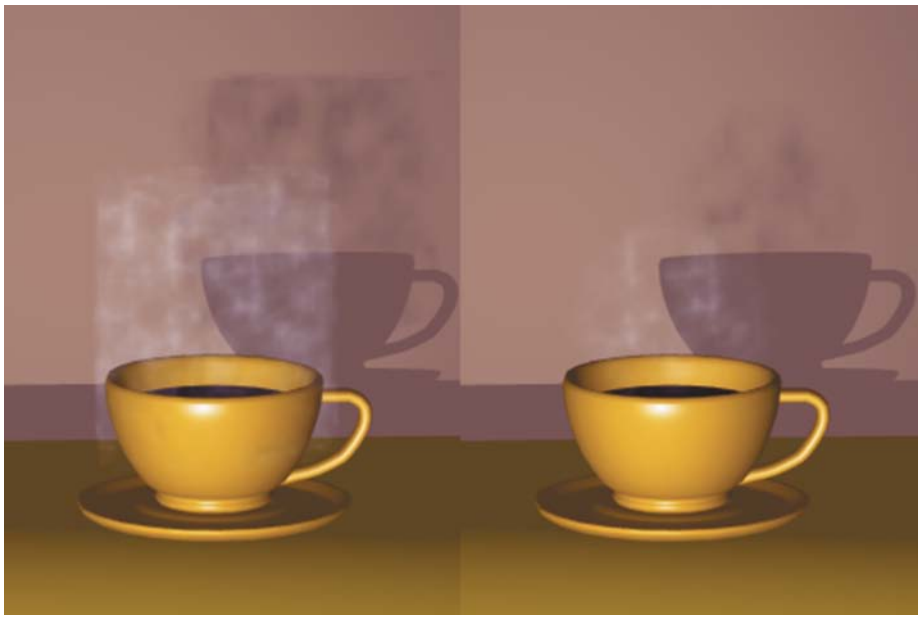


FIGURE 7.2 Preliminary steam rising from a teacup. Left: No shaping of the steam. Right: Only spherical attenuation. Copyright © 1992 David S. Ebert.

```
{
 float     turb, dist_sq,density_max;
   int
i, indx;
xyz_td diff;
static float pow_table[POW_TABLE_SIZE], ramp[RAMP_SIZE],
            offset[OFFSET_SIZE];
static int calcd=1;
if(calcd) { calcd=0;
   for(i=POW_TABLE_SIZE-1; i>=0; i--)
     pow_table[i] =
        (float)pow(((double)(i))/(POW_TABLE_SIZE-1)*
                   parms[1]* 2.0,(double)parms[2]);
   make_tables(ramp);
  }
turb = fast_turbulence(pnt, pixel_size);
*density = pow_table[(int)(turb*0.5*(POW_TABLE_SIZE-1))];

 // determine distance from center of the slab ^2.
 XYZ_SUB(diff,vol.shape.center, pnt_world);
 dist_sq = DOT_XYZ(diff,diff);
```

```
 density_max = dist_sq*vol.shape.inv_rad_sq.y;
 indx = (int) ((pnt.x+pnt.y+pnt.z)*100) & (OFFSET_SIZE -1);
 density_max += parms[3]*offset[indx];

  if(density_max >= .25) // ramp off if > 25% from center
  { // get table index 0:RAMP_SIZE-1
    i = (density_max -.25)*4/3*RAMP_SIZE;
    i=MIN(i,RAMP_SIZE-1);
    density_max = ramp[i];
    *density *=density_max;
  }
}

void make_tables(float *ramp, float *offset)
{
  int i;
  float dist;
  srand48(42);
  for(i=0; i < OFFSET_SIZE; i++)
   {
    offset[i] = (float)drand48( );
   }
 for(i = 0; i < RAMP_SIZE; i++)
  { dist =i/(RAMP_SIZE -1.0);
    ramp[i]=(cos(dist*M_PI) +1.0)/2.0;
   }
}
```

These modifications produce the more realistic image seen in Figure 7.2 (right). Two additional parameters are used in this new procedure: `pnt_world` and `vol`. `pnt_world` is the location of the point in world space; `vol` is a structure containing information on the volume being rendered. Table 7.1 clarifies the use of the various variables.

The procedure now ramps off the density spherically using a cosine falloff function. If the distance from the center squared is greater than 25%, the cosine falloff is applied. The resulting image can be seen on the right in Figure 7.2. This image is better than the one shown on the left in Figure 7.2, but still lacking.

To solve the second problem, the gas density decreasing as it rises, the density of the gas needs to be ramped off as it rises to get a more natural look. The following addition to the end of the `steam_slab1` procedure will accomplish this:

```
dist = pnt_world.y - vol.shape.center.y;
if(dist > 0.0)
  { dist = (dist +offset[indx]*.1)*vol.shape.inv_rad.y;
    if(dist > .05)
```

TABLE 7.1 VARIABLES FOR STEAM PROCEDURE

VARIABLE	DESCRIPTION
`pnt`	location of the point in the solid texture space
`pnt_world`	location of the point in world space
`density`	the value returned from the function
`parms[1]`	maximum density of the gas
`parms[2]`	exponent for the power function for gas shaping
`parms[3]`	amount of randomness to use in falloff
`parms[4]`	distance at which to start ramping off the gas density
`vol.shape.center`	center of the volume
`vol.shape.inv_rad_sq`	1/radius squared of the slab
`dist_sq`	point's distance squared from the center of the volume
`density_max`	density scaling factor based on distance squared from the center
`indx`	an index into a random number table
`offset`	a precomputed table of random numbers used to add noise to the ramp off of the density
`ramp`	a table used for cosine falloff of the density values

```
    { offset2 = (dist -.05)*1.111111;
     offset2 = 1 - (exp(offset2)-1.0)/1.718282;
     offset2 *= parms[1];
     *density *= offset;
    }
}
```

This procedure uses the e^x function to decrease the density as the gas rises. If the vertical distance above the center is greater than 5% of the total distance, the density is exponentially ramped off to 0. The result of this addition to the above procedure can be seen in Figure 7.3. As can be seen in this image, the resulting steam is very convincing. In the next chapter, animation effects using this basic steam model will be presented.

A Single Column of Smoke

The final example procedure creates a single column of rising smoke. The basis of the smoke shape is a vertical cylinder. Two observations can make the resulting

FIGURE 7.3 Final image of steam rising from a teacup, with both spherical and height density attenuation. Copyright © 1997 David S. Ebert.

image look realistic. First, smoke disperses as it rises. Second, the smoke column is initially fairly smooth, but as the smoke rises, turbulent behavior becomes the dominant characteristic of the flow. In order to reproduce these observations, turbulence is added to the cylinder's center to make the column of smoke look more natural. To simulate air currents and general turbulent effects, more turbulence is added as the height from the bottom of the smoke column increases. To simulate dispersion, the density of the gas is ramped off to zero as the gas rises. These ideas will produce a very straight column of smoke. The following additional observation will make the image more realistic: smoke tends to bend and swirl as it rises. Displacing each point by a vertical spiral (helix) creates the swirling of the smoke. The x- and z-coordinates of the point are displaced by the cosine and sine of the angle of rotation. The y-coordinate of the point is displaced by the turbulence of the point. The following procedure produces a single column of smoke based on these observations.

```
// ////////////////////////////////////////////////////////////
//                    Smoke_stream
// ////////////////////////////////////////////////////////////
// parms[1] = Maximum density value - density scaling factor
// parms[2] = height for 0 density (end of ramping it off)
```

```
// parms[3] = height to start adding turbulence
// parms[4] = height(length) for maximum turbulence;
// parms[5] = height to start ramping density off
// parms[6] = center.y
// parms[7] = speed for rising
// parms[8] = radius
// parms[9] = max radius of swirling
// ////////////////////////////////////////////////////////////

void smoke_stream(xyz_td pnt, float *density, float *parms,
                  xyz_td pnt_world, vol_td *vol)
{
 float        dist_sq;
 extern float offset[OFFSET_SIZE];
 xyz_td       diff;
 xyz_td       hel_path, new_path, direction2, center;
 double       ease( ), turb_amount, theta_swirl, cos_theta,
              sin_theta;
 static int   calcd=1;
 static float cos_theta2, sin_theta2;
static xyz_td bottom;
static double rad_sq, max_turb_length, radius, big_radius,
              st_d_ramp, d_ramp_length, end_d_ramp,
              inv_max_turb_length;
double        height, fast_turb, t_ease, path_turb, rad_sq2;
if(calcd)
  { bottom.x = 0; bottom.z = 0;
   bottom.y = parms[6];
   radius   = parms[8];
   big_radius = parms[9];
   rad_sq = radius*radius;
   max_turb_length = parms[4];
   inv_max_turb_length = 1/max_turb_length;
   st_d_ramp = parms[5];
   end_d_ramp = parms[2];
   d_ramp_length = end_d_ramp - st_d_ramp;
   theta_swirl = 45.0*M_PI/180.0; // swirling effect
   cos_theta = cos(theta_swirl);
   sin_theta = sin(theta_swirl);
   cos_theta2 = .01*cos_theta;
   sin_theta2 = .0075*sin_theta;
   calcd=0;
  }

height = pnt_world.y - bottom.y + fast_noise(pnt)*radius;
// We don't want smoke below the bottom of the column
if(height < 0)
  { *density =0; return;}
height -= parms[3];
```

```
if (height < 0.0)
    height =0.0;
// calculate the eased turbulence, taking into account the value
// may be greater than 1, which ease won't handle.
t_ease = height* inv_max_turb_length;
if(t_ease > 1.0)
  { t_ease = ((int)(t_ease)) +ease( (t_ease - ((int)t_ease)),
    .001, .999);
    if( t_ease > 2.5)
        t_ease = 2.5;
  }
else
  t_ease = ease(t_ease, .5, .999);
// Calculate the amount of turbulence to add in
fast_turb= fast_turbulence(pnt);
turb_amount = (fast_turb -0.875)* (.2 + .8*t_ease);
path_turb = fast_turb*(.2 + .8*t_ease);
// add turbulence to the height and see if it is above the top
height +=0.1*turb_amount;
if(height > end_d_ramp)
   { *density=0; return; }
//increase the radius of the column as the smoke rises
if(height <=0)
   rad_sq2 = rad_sq*.25;
else if (height <=end_d_ramp)
   { rad_sq2 = (.5 + .5*(ease( height/(1.75*end_d_ramp), .5,
               .5)))*radius;
     rad_sq2 *=rad_sq2;
   }
// ***************************************************
// move along a helical path
// ***************************************************

// calculate the path based on the unperturbed flow: helical path

hel_path.x = cos_theta2 *(1+ path_turb)*
           (1+cos(pnt_world.y*M_PI*2) *.11)
           *(1+ t_ease*.1) + big_radius*path_turb;
hel_path.z = sin_theta2 *(1+path_turb)*
           (1+sin(pnt_world.y*M_PI*2)*.085)*
           (1+ t_ease*.1) + .03*path_turb;
hel_path.y = - path_turb;
XYZ_ADD(direction2, pnt_world, hel_path);

//adjusting the center point for ramping off the density based on
//the turbulence of the moved point
turb_amount *= big_radius;
center.x = bottom.x - turb_amount;
center.z = bottom.z + .75*turb_amount;
//calculate the radial distance from the center and ramp off the
```

Team LRN

FIGURE 7.4 A rising column of smoke. Copyright © 1994 David S. Ebert.

```
// density based on this distance squared.
diff.x = center.x - direction2.x;
diff.z = center.z - direction2.z;
dist_sq = diff.x*diff.x + diff.z*diff.z;
if(dist_sq > rad_sq2)
  {*density=0; return;}
*density = (1-dist_sq/rad_sq2 + fast_turb*.05) *
           parms[1];
if(height > st_d_ramp)
   *density *= (1- ease((height - st_d_ramp)/(d_ramp_length), .5, .5));
}
```

The result of this procedure can be seen in Figure 7.4. In this procedure, turbulence is added to many variables before doing tests and computing paths. The addition of the turbulence produces a more natural appearance to the column of smoke. This procedure uses the same `offset` table as in the `steam_slab1` procedure. An `ease` procedure is also used to perform ease-in and ease-out of the turbulence addition and density ramping. The helical path that is used is actually the multiplication of two helical paths with the addition of turbulence. This calculation provides better

TABLE 7.2 PARAMETERS FOR SMOKE COLUMN PROCEDURE

parm	VALUE	DESCRIPTION
1	0.93	density scaling factor
2	1.6	height for 0 density (end of ramping it off)
3	0.175	height to start adding turbulence
4	0.685	height for maximum turbulence
5	0.0	height to start ramping density off
6	−0.88	`center.y`
7	2.0	speed for rising
8	0.04	radius
9	0.08	maximum radius of swirling

results than a single helical path. The parameter values and their description can be found in Table 7.2.

CONCLUSION

This chapter has provided an introduction to the solid space framework for procedural modeling and texturing and shown several example procedures for producing still images of gases. Animating these procedures is the topic of the next chapter, while Chapter 10 discusses methods for accelerating these techniques using graphics hardware.

Team LRN

8

ANIMATING SOLID SPACES

DAVID S. EBERT

The previous chapter discussed modeling the geometry of gases. This chapter discusses animating gases and other procedurally defined solid spaces. There are several ways that solid spaces can be animated. This chapter will consider two approaches:

1. Changing the solid space over time
2. Moving the point being rendered through the solid space

The first approach has time as a parameter that changes the definition of the space over time, a very natural and obvious way to animate procedural techniques. With this approach, time has to be considered in the design of the procedure, and the procedure evolves with time. Procedures that change the space to simulate growth, evolution, or aging are common examples of this approach. A related technique creates the procedure in a four-dimensional space, with time as the fourth dimension, such as a 4D noise function.

The second approach does not actually change the solid space, but moves the point in the volume or object over time through the space, in effect procedurally warping or perturbing the space. Moving the fixed three-dimensional screen space point along a path over time through the solid space before evaluating the turbulence function animates the gas (solid texture, hypertexture). Each three-dimensional screen space point is inversely mapped back to world space. From world space, it is mapped into the gas and turbulence space through the use of simple affine transformations. Finally, it is moved through the turbulence space over time to create the movement. Therefore, the path direction will have the reverse visual effect. For example, a *downward* path applied to the screen space point will show the texture or volume object *rising*. Figure 8.1 illustrates this process.

Both of these techniques can be applied to solid texturing, gases, and hypertextures. After a brief discussion of animation paths, the application of these two

Team LRN

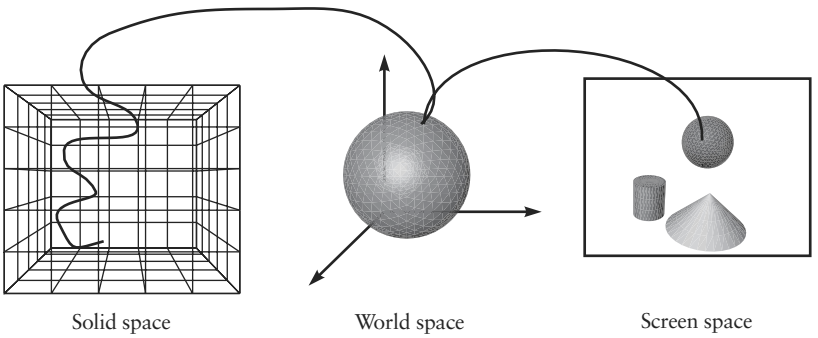


Figure 8.1 Moving a screen space point through the solid space.

techniques to solid texturing is discussed, followed by an exploration of the way they are used for gas animation and hypertextures, including liquids. Finally, the chapter concludes with a discussion of an additional procedural animation technique, particle systems.

ANIMATION PATHS

This chapter will describe various ways of creating animation paths for movement through the solid space. For many examples, I will use a helical (spiral) path. There are two reasons for using helical paths. First, most gases do not move along a linear path. Turbulence, convection, wind, and so on, change the path movement. From my observations, smoke, steam, and fog tend to swirl while moving in a given direction. A helical path can capture this general sense of motion. Second, helical paths are very simple to calculate. The calculation involves rotation around the axis of the helix (direction of motion) and movement along the axis. To create the rotation, the sine and cosine functions are used. The angle for these functions is based on the frame number to produce the rotation over time. The rate of rotation can be controlled by taking the frame number modulo a constant. Linear motion, again based on the frame number, will be used to create the movement along the axis.

The code segment below creates a helical path that rotates about the axis once every 100 frames. The speed of movement along the axis is controlled by the variable `linear_speed`.

```
theta  = (frame_number%100)*(2*M_PI/100);
path.x = cos(theta);
path.y = sin(theta);
path.z = theta*linear_speed;
```

One final point will clarify the procedures given in this chapter. To get smooth transitions between values and smooth acceleration and deceleration, *ease-in* and *ease-out* procedures are used. These are the standard routines used by animators to stop a moving object from jumping instantaneously from a speed of 0 to a constant velocity. One simple implementation of these functions assumes a sine curve for the acceleration and integrates this curve over one-half of its period.

ANIMATING SOLID TEXTURES

This section will show how the previous two animation approaches can be used for solid texturing. Applying these techniques to color solid texturing will be discussed first, followed by solid textured transparency.

A marble procedure will be used as an example of color solid texture animation. The following simple marble procedure is based on Perlin's `marble` function (Perlin 1985). An interactive hardware-accelerated version of `marble` is described in Chapter 10.

```
rgb_td marble(xyz_td pnt)
{
  float y;
  y = pnt.y + 3.0*turbulence(pnt, .0125);
  y = sin(y*M_PI);
  return (marble_color(y));
}
rgb_td marble_color(float x)
{
  rgb_td clr;
  x = sqrt(x+1.0)*.7071;
  clr.g = .30 + .8*x;
  x=sqrt(x);
  clr.r = .30 + .6*x;
  clr.b = .60 + .4*x;
  return (clr);
}
```

This procedure applies a sine function to the turbulence of the point. The resulting value is then mapped to the color. The results achievable by this procedure can be seen in Figure 8.2 (lower right).

Marble Forming

The application of the previous two animation approaches to this function has very different effects. When the first approach is used, changing the solid space over time,

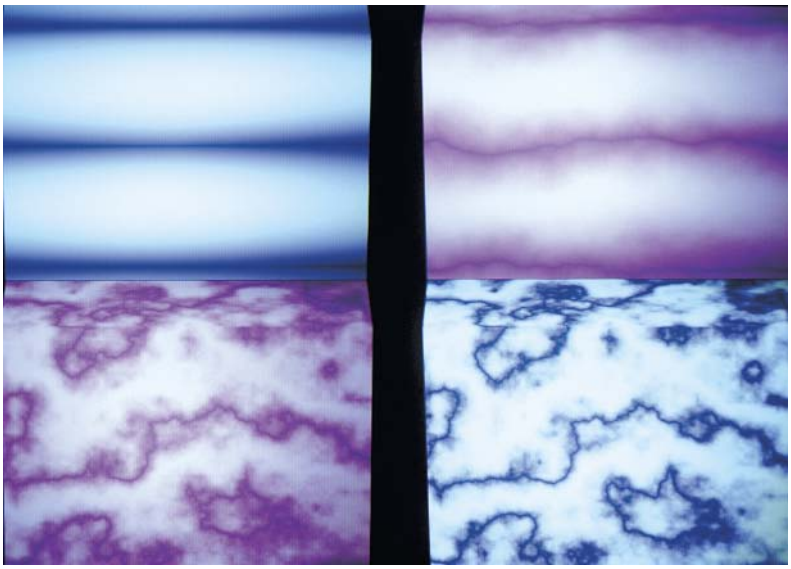


FIGURE 8.2 Marble forming. The images show the banded material heating, deforming, then cooling and solidifying. Copyright © 1992 David S. Ebert.

the formation of marble from banded rock can be achieved. Marble is formed from the turbulent mixing of different bands of rock. To simulate this process, initially no turbulence is added to the point; therefore, the sine function determines the color. Basing the color on the sine function produces banded material. As the frame number increases, the amount of turbulence added to the point is increased, deforming the bands into the marble vein pattern. The resulting procedure is the following:

```
rgb_td marble_forming(xyz_td pnt, int frame_num, int
                      start_frame, int end_frame)
 {
 float x, turb_percent, displacement;

 if(frame_num < start_frame)
    { turb_percent=0;
      displacement=0;
    }
   else if (frame_num >= end_frame)
    { turb_percent=1;
      displacement=3;
    }
   else
    { turb_percent= ((float)(frame_num-start_frame))/
```

```
                    (end_frame-start_frame);
      displacement = 3*turb_percent;
    }
x = pnt.x + turb_percent*3.0*turbulence(pnt, .0125) -
    displacement;
x = sin(x*M_PI);
return (marble_color(x));
}
```

The `displacement` value in this procedure is used to stop the entire texture from moving. Without the `displacement` value, the entire banded pattern moves horizontally to the left of the image, instead of the veins forming in place.

This procedure produces the desired effect, but the realism of the results can be increased by a few small changes. First of all, ease-in and ease-out of the rate of adding turbulence will give more natural motion. Second, the color of the marble can be changed to simulate heating before and during the deformation, and to simulate cooling after the deformation. The marble color is blended with a "glowing" marble color to simulate the heating and cooling. (Even though this may not be physically accurate, it produces a nice effect.) This can be achieved by the following procedure:

```
rgb_td marble_forming2(xyz_td pnt, int frame_num, int start_frame,
                       int end_frame, int heat_length)
 {
  float x, turb_percent, displacement, glow_percent;
  rgb_td m_color;
  if(frame_num < (start_frame-heat_length/2) ||
     frame_num > end_frame+heat_length/2)
      glow_percent=0;
  else if (frame_num < start_frame + heat_length/2)
      glow_percent= 1.0 - ease( ((start_frame+heat_length/2-
                          frame_num)/ heat_length),0.4, 0.6);
  else if (frame_num > end_frame-heat_length/2)
      glow_percent = ease( ((frame_num-(end_frame-
                       heat_length/2))/heat_length),0.4, 0.6);
               else
      glow_percent=1.0;

  if(frame_num < start_frame)
    { turb_percent=0; displacement=0;
    }
  else if (frame_num >= end_frame)
    { turb_percent=1; displacement=3;
    }
  else
    { turb_percent= ((float)(frame_num-start_frame))/
                       (end_frame-start_frame);
      turb_percent=ease(turb_percent, 0.3, 0.7);
```

```
      displacement = 3*turb_percent;
    }
  x = pnt.y + turb_percent*3.0*turbulence(pnt, .0125) -
      displacement;
  x = sin(x*M_PI);
  m_color=marble_color(x);
  glow_percent= .5* glow_percent;
  m_color.r= glow_percent*(1.0)+ (1-glow_percent)*m_color.r;
  m_color.g= glow_percent*(0.4)+ (1-glow_percent)*m_color.g;
  m_color.b= glow_percent*(0.8)+ (1-glow_percent)*m_color.b;
  return(m_color);
}
```

The resulting images can be seen in Figure 8.2. This figure shows four images of the change in the marble from banded rock (upper-left image) to the final marbled rock (lower-right image). Of course, the resulting sequence would be even more realistic if the material actually deformed, instead of the color simply changing. This effect will be described in the "Animating Hypertextures" section.

Marble Moving

A different effect can be achieved by the second animation approach, moving the point through the solid space. Any path can be used for movement through the marble space. A simple, obvious choice would be a linear path. Another choice, which produces very ethereal patterns in the material, is to use a turbulent path. The procedure below uses yet another choice for the path. This procedure moves the point along a horizontal helical path before evaluating the turbulence function, producing the effect of the marble pattern moving through the object. The helical path provides a more interesting result than the linear path, but does not change the general marble patterns as does using a turbulent path through the turbulence space. This technique can be used to determine the portion of marble from which to "cut" the object in order to achieve the most pleasing vein patterns. (You are in essence moving the object through a three-dimensional volume of marble.)

```
rgb_td moving_marble(xyz_td pnt, int frame_num)
 {
 float        x, tmp, tmp2;
 static float down, theta, sin_theta, cos_theta;
 xyz_td       hel_path, direction;
 static int   calcd=1;

 if(calcd)
    { theta=(frame_num%SWIRL_FRAMES)*SWIRL_AMOUNT;//swirling
      cos_theta = RAD1 * cos(theta) + 0.5;
```

```
      sin_theta = RAD2 * sin(theta) - 2.0;
      down = (float)frame_num*DOWN_AMOUNT+2.0;
      calcd=0;
    }
  tmp = fast_noise(pnt); // add some randomness
  tmp2 = tmp*1.75;
// calculate the helical path
  hel_path.y = cos_theta + tmp;
  hel_path.x = (-down) + tmp2;
  hel_path.z = sin_theta - tmp2;
  XYZ_ADD(direction, pnt, hel_path);
  x = pnt.y + 3.0*turbulence(direction, .0125);
  x = sin(x*M_PI);
  return (marble_color(x));
 }
```

In this procedure, `SWIRL_FRAMES` and `SWIRL_AMOUNT` determine the number of frames for one complete rotation of the helical path. By choosing `SWIRL_FRAMES` = 126 and `SWIRL_AMOUNT` = $2\pi/126$, the path swirls every 126 frames. `DOWN_AMOUNT` controls the speed of the downward movement along the helical path. A reasonable speed for downward movement for a unit-sized object is to use `DOWN_AMOUNT` = 0.0095. `RAD1` and `RAD2` are the y and z radii of the helical path.

Animating Solid Textured Transparency

This section describes the use of the second solid space animation technique, moving the point through the solid space, for animating solid textured transparency.

This animation technique is the one that I originally used for animating gases and is still the main technique that I use for gases. The results of this technique applied to solid textured transparency can be seen in Ebert, Boyer, and Roble (1989). The `fog` procedure given next is similar in its animation approach to the earlier `moving_marble` procedure. It produces fog moving through the surface of an object and can be used as a surface-based approach to simulate fog or clouds. Again in this procedure, a downward helical path is used for the movement through the space, which produces an upward swirling to the gas movement.

```
void fog(xyz_td pnt, float *transp, int frame_num)
{
 float tmp;
 xyz_td direction,cyl;
 double theta;
 pnt.x += 2.0 +turbulence(pnt, .1);
 tmp = noise_it(pnt);
 pnt.y += 4+tmp; pnt.z += -2 - tmp;
```

Figure 8.3 Solid textured transparency-based fog. Copyright © 1994 David S. Ebert.

```
theta =(frame_num%SWIRL_FRAMES)*SWIRL_AMOUNT;
 cyl.x =RAD1 * cos(theta); cyl.z =RAD2 * sin(theta);
 direction.x   = pnt.x + cyl.x;
 direction.y   = pnt.y - frame_num*DOWN_AMOUNT;
 direction.z   = pnt.z + cyl.z;
 *transp = turbulence(direction, .015);
 *transp = (1.0 -(*transp)*(*transp)*.275);
 *transp =(*transp)*(*transp)*(*transp);
}
```

An image showing this procedure applied to a cube can be seen in Figure 8.3. The values used for this image can be found in Table 8.1.

Another example of the use of solid textured transparency animation can be seen in Figure 8.4, which contains a still from an animation entitled *Once a Pawn a Foggy Knight . . .* (Ebert, Boyer, and Roble 1989). In this scene, three planes are positioned to give a two-dimensional approximation of three-dimensional fog. One plane is in front of the scene, one plane is approximately in the middle, and the final plane is behind all the objects in the scene.

This technique is similar to Gardner's technique for producing images of clouds (Gardner 1985), except that it uses turbulence to control the transparency instead of Fourier synthesis. As with any surface-based approach to modeling gases, including

Team LRN

TABLE 8.1 VALUES FOR FOG PROCEDURE

PARAMETER	VALUE
DOWN_AMOUNT	0.0095
SWIRL_FRAMES	126
SWIRL_AMOUNT	$2\pi/126$
RAD1	0.12
RAD2	0.08

FIGURE 8.4 A scene from *Once a Pawn a Foggy Knight . . .* showing solid textured transparency used to simulate fog. Copyright © 1989 David S. Ebert.

Gardner's, this technique cannot produce three-dimensional volumes of fog or accurate shadowing from the fog.

ANIMATION OF GASEOUS VOLUMES

As described in the previous section, animation technique 2, moving the point through the solid space, is the technique that I use to animate gases. This technique

will be used in all the examples in this section. Moving each fixed three-dimensional screen space point along a path over time through the solid space before evaluating the turbulence function creates the gas movement. First, each three-dimensional screen space point is inversely mapped back to world space. Second, it is mapped from world space into the gas and turbulence space through the use of simple affine transformations. Finally, it is moved through the turbulence space over time to create the movement of the gas. Therefore, the path direction will have the reverse visual effect. For example, a downward path applied to the screen space point will cause the gas to rise.

This gas animation technique can be considered to be the inverse of particle systems because each point in three-dimensional screen space is moved through the gas space to see which portion of the gas occupies the current location in screen space. The main advantage of this approach over particle systems is that extremely large geometric databases of particles are not required to get realistic images. The complexity is always controlled by the number of screen space points in which the gas is potentially visible.

Several interesting animation effects can be achieved through the use of helical paths for movement through the solid space. These helical path effects will be described first, followed by the use of three-dimensional tables for controlling the gas movement. Finally, several additional primitives for creating gas animation will be presented.

Helical Path Effects

Helical paths can be used to create several different animation effects for gases. In this chapter, three examples of helical path effects will be presented: steam rising from a teacup, rolling fog, and a rising column of smoke.

Steam Rising from a Teacup

In the previous chapter, a procedure for producing a still image of steam rising from a teacup was described. This procedure can be modified to produce convincing animations of steam rising from the teacup by the addition of helical paths for motion. Each point in the volume is moved downward along a helical path to produce the steam rising and swirling in the opposite direction. The modification needed is given below. This animation technique is the same technique that was used in the `moving_marble` procedure.

```
    void steam_moving(xyz_td pnt, xyz_td pnt_world, float *density,
                      float *parms, vol_td vol)
    {
*** float noise_amt,turb, dist_sq, density_max, offset2, theta, dist;
    static float pow_table[POW_TABLE_SIZE], ramp[RAMP_SIZE],
                 offset[OFFSET_SIZE];
    extern int frame_num;
    xyz_td direction, diff;
    int i, indx;
    static int calcd=1;
*** static float down, cos_theta, sin_theta;

    if(calcd)
      { calcd=0;
       // determine how to move point through space(helical path)
***    theta =(frame_num%SWIRL_FRAMES)*SWIRL;
***    down = (float)frame_num*DOWN*3.0 +4.0;
***    cos_theta = RAD1*cos(theta) +2.0;
***    sin_theta = RAD2*sin(theta) -2.0;
       for(i=POW_TABLE_SIZE-1; i>=0; i--)
          pow_table[i] =(float)pow(((double)(i))/(POW_TABLE_SIZE-1)*
                        parms[1]* 2.0,(double)parms[2]);
       make_tables(ramp);
       }
    // move the point along the helical path
*** noise_amt = fast_noise(pnt);
*** direction.x = pnt.x + cos_theta + noise_amt;
*** direction.y = pnt.y - down + noise_amt;
*** direction.z = pnt.z +sin_theta + noise_amt;
    turb =fast_turbulence(direction);
    *density = pow_table[(int)(turb*0.5*(POW_TABLE_SIZE-1))];
    // determine distance from center of the slab ^2.
    XYZ_SUB(diff,vol.shape.center, pnt_world);
    dist_sq = DOT_XYZ(diff,diff) ;
    density_max = dist_sq*vol.shape.inv_rad_sq.y;
    indx = (int)((pnt.x+pnt.y+pnt.z)*100) & (OFFSET_SIZE -1);
    density_max += parms[3]*offset[indx];
    if(density_max >= .25) // ramp off if > 25% from center
      { // get table index 0:RAMP_SIZE-1
       i = (density_max -.25)*4/3*RAMP_SIZE;
       i=MIN(i,RAMP_SIZE-1);
       density_max = ramp[i];
       *density *=density_max;
      }
    // ramp it off vertically
    dist = pnt_world.y - vol.shape.center.y;
    if(dist > 0.0)
     { dist = (dist +offset[indx]*.1)*vol.shape.inv_rad.y;
```

```
    if(dist > .05)
      { offset2 = (dist -.05)*1.111111;
       offset2 = 1 - (exp(offset2)-1.0)71.718282;
       offset2*=parms[1];
       *density *= offset2;
      }
   }
  }
```

The lines that have changed from the earlier `steam_slab1` procedure are marked with three asterisks (`***`). This procedure creates upward swirling movement in the gas, which swirls around 360 degrees every `SWIRL_FRAMES` frame. Noise is applied to the path to make it appear more random. The parameters `RAD1` and `RAD2` determine the elliptical shape of the swirling path. Additional variables in this procedure are the angle of rotation about the helical path (`theta`), the frame number (`frame_num`), the cosine of the angle of rotation (`cos_theta`), the sine of the angle of rotation (`sin_theta`), the amount to move along the helical axis (`down`), a noise amount to add to the path (`noise_amt`), and the new location of the point after movement along the path (`direction`).

The downward helical path through the gas space produces the effect of the gas rising and swirling in the opposite direction.

For more realistic steam motion, a simulation of air currents is helpful. Adding turbulence to the helical path can approximate this, where the amount of turbulence added is proportional to the height above the teacup. (This assumes that no turbulence is added at the surface.)

Fog Animation

The next example of helical path effects is the creation of rolling fog. For this animation, a horizontal helical path will be used to create the swirling motion of the fog to the right of the scene. From examining the following `volume_fog_animation` procedure, it is clear that this procedure uses the same animation technique as the earlier `steam_moving` procedure: move each point along a helical path before evaluating the turbulence function. The value returned by the turbulence function is again multiplied by a density scalar factor, `parms[1]`, and raised to a power, `parms[2]`. As in the previous procedures, a precomputed table of density values raised to a power is used to speed calculation. A more complete description of the use of helical paths for producing fog animation can be found in Ebert and Parent (1990).

```
void volume_fog_animation(xyz_td pnt, xyz_td pnt_world, float
                          *density, float *parms, vol_td vol)
```

```
{
 float noise_amt, turb;
 extern int frame_num;
 xyz_td direction;
 int indx;
 static float pow_table[POW_TABLE_SIZE];
 int i;
 static int calcd=1;
 static float down, cos_theta, sin_theta, theta;
 if(calcd)
   {
    down = (float)frame_num*SPEED*1.5 +2.0;
    theta =(frame_num%SWIRL_FRAMES)*SWIRL_AMOUNT;//get swirling effect
    cos_theta = cos(theta)*.1 + 0.5; //use a radius of .1
    sin_theta = sin(theta)*.14 - 2.0; //use a radius of .14
    calcd=0;
    for(i=POW_TABLE_SIZE-1; i>=0; i--)
     {
      pow_table[i]=(float)pow(((double)(i))/(POW_TABLE_SIZE-1)*
                parms[1]*4.0,(double)parms[2]);
     }
   }
 // make it move horizontally & add some noise to the movement
 noise_amt = fast_noise(pnt);
 direction.x = pnt.x - down + noise_amt*1.5;
 direction.y = pnt.y + cos_theta +noise_amt;
 direction.z = pnt.z + sin_theta -noise_amt*1.5;
 // base the turbulence on the new point
 turb =fast_turbulence(direction);
 *density = pow_table[(int)((turb*turb)*(.25*(POW_TABLE_SIZE-1)))];
 // make sure density isn't greater than 1
 if(*density >1)
   *density=1;
}
```

As in the `fog` and `steam_moving` procedures, the `volume_fog_animation` procedure uses the same values for `SWIRL_FRAMES` (126) and `SWIRL_AMOUNT` ($2\pi/126$). `SPEED` controls the rate of horizontal movement, and the value I use to produce gently rolling fog is 0.012. The results achievable by this procedure can be seen in Figure 8.5, which is a still from an animation entitled *Getting into Art* (Ebert, Ebert, and Boyer 1990). For this image, `parms[1] = 0.22` and `parms[2] = 4.0`.

Smoke Rising

The final example of helical path effects is the animation of the `smoke_stream` procedure given earlier to create a single column of smoke. Two different helical paths are

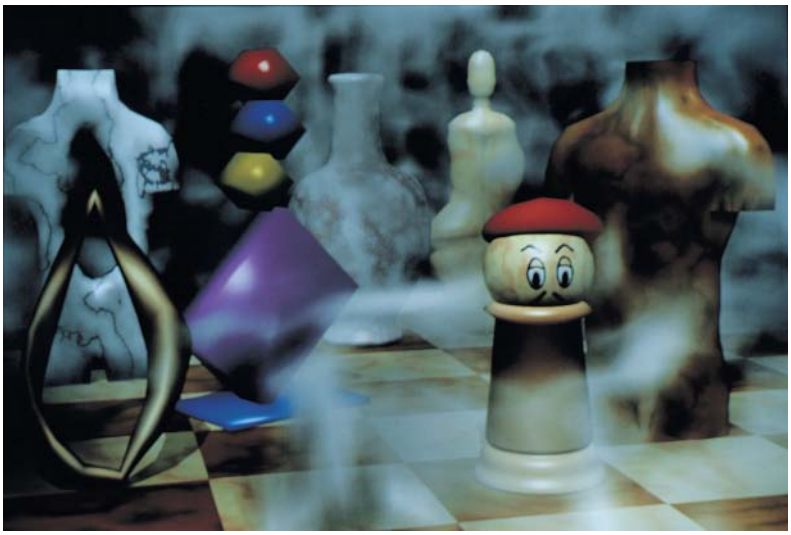


FIGURE 8.5 A scene from *Getting into Art*, showing volume-rendered fog animation created by horizontal helical paths. Copyright © 1990 David S. Ebert.

used to produce the swirling column of smoke. This `smoke_stream` procedure already used a helical path to displace each point to get a more convincing column of smoke. We will now modify this helical path to make it a downward helical path based on the frame number, creating the rising column of smoke. The second helical path will actually displace the center point of the cylinder, producing a swirling cylinder of smoke (instead of a vertical cylinder as was used in Chapter 7). This second helical path will swirl at a different rate than the first. The same input parameter values can be used for this procedure. The following is the procedure that is the result of these modifications.

```
// ********************************************************
//  Rising_smoke_stream
// ********************************************************
// parms[1] = maximum density value - density scaling factor
// parms[2] = height for 0 density (end of ramping it off)
// parms[3] = height to start adding turbulence
// parms[4] = height (length) for maximum turbulence
// parms[5] = height to start ramping off density
// parms[6] = center.y
// parms[7] = speed for rising
// parms[8] = radius
```

```
// parms[9] = max radius of swirling
// *******************************************************
void rising_smoke_stream(xyz_td pnt,float *density, float
                         *parms, xyz_td pnt_world, vol_td *vol)
{
 float dist_sq;
 extern float offset[OFFSET_SIZE];
 extern int frame_num;
 static int calcd=1;
 static float down, cos_theta2, sin_theta2;
 xyz_td hel_path, center, diff, direction2;
 double ease(), turb_amount, theta_swirl, cos_theta, sin_theta;
 static xyz_td bottom;
 static double rad_sq, max_turb_length, radius, big_radius,
               st_d_ramp, d_ramp_length, end_d_ramp, down3,
               inv_max_turb_length, cos_theta3, sin_theta3;
 double        height, fast_turb, t_ease, path_turb, rad_sq2;

 if(calcd)
   {
    bottom.x = 0; bottom.z = 0;
    bottom.y = parms[6];
    radius = parms[8];
    big_radius = parms[9];
    rad_sq = radius*radius;
    max_turb_length = parms[4];
    inv_max_turb_length = 1/max_turb_length;
    st_d_ramp = parms[5];
    st_d_ramp =MIN(st_d_ramp, end_d_ramp);
    end_d_ramp = parms[2];
    d_ramp_length = end_d_ramp - st_d_ramp;
    //calculate rotation about the helix axis based on frame_number

*** theta_swirl=(frame_num%SWIRL_FRAMES_SMOKE)*SWIRL_SMOKE; // swirling
*** cos_theta = cos(theta_swirl);
*** sin_theta = sin(theta_swirl);
*** down = (float)(frame_num)*DOWN_SMOKE*.75 * parms[7];
    // Calculate sine and cosine of the different radii of the
    // two helical paths
*** cos_theta2 = .01*cos_theta;
*** sin_theta2 = .0075*sin_theta;
*** cos_theta3= cos_theta2*2.25;
*** sin_theta3= sin_theta2*4.5;
*** down3= down*2.25;
    calcd=0;
   }
  height = pnt_world.y - bottom.y + fast_noise(pnt)*radius;
  // We don't want smoke below the bottom of the column
  if(height < 0)
```

```
    { *density =0; return;}
  height -= parms[3];
  if (height < 0.0)
    height =0.0;
  // calculate the eased turbulence, taking into account the
  // value may be greater than 1, which ease won't handle.
  t_ease = height* inv_max_turb_length;
  if(t_ease > 1.0)
    { t_ease =((int)(t_ease))+ease((t_ease-((int)t_ease)), .001,.999);
      if( t_ease > 2.5) t_ease = 2.5;
    }
  else
    t_ease = ease(t_ease, .5, .999);
  // move point along the helical path before evaluating turbulence
*** pnt.x += cos_theta3;
*** pnt.y -= down3;
*** pnt.z += sin_theta3;
    fast_turb= fast_turbulence_three(pnt);
    turb_amount = (fast_turb -0.875)* (.2 + .8*t_ease);
    path_turb = fast_turb*(.2 + .8*t_ease);
    // add turbulence to the height & see if it is above the top
    height +=0.1*turb_amount;
    if(height > end_d_ramp)
      { *density=0; return; }
    // increase the radius of the column as the smoke rises
    if(height <=0)
      rad_sq2 = rad_sq*.25;
    else if (height <=end_d_ramp)
      {
       rad_sq2 =(.5 +.5*(ease( height/(1.75*end_d_ramp),.5, .5)))*radius;
       rad_sq2 *=rad_sq2;
      }
    else
       rad_sq2 = rad_sq;
    //
    // move along a helical path plus add the ability to use tables
    //
    // calculate the path based on the unperturbed flow: helical path
    //
*** hel_path.x = cos_theta2 *(1+path_turb)*(1+t_ease*.1)*
       (1+cos((pnt_world.y+down*.5)*M_PI*2)*.11) + big_radius*path_turb;
*** hel_path.z = sin_theta2 * (1+path_turb)*(1+ t_ease*.1)*
       (1+sin((pnt_world.y +down*.5)*M_PI*2)*.085) + .03*path_turb;
*** hel_path.y = (- down) - path_turb;
    XYZ_ADD(direction2, pnt_world, hel_path);
    // adjusting center point for ramping off density based on the
    // turbulence of the moved point
   turb_amount *= big_radius;
   center.x = bottom.x - turb_amount;
   center.z = bottom.z + .75*turb_amount;
```

```
    // calculate the radial distance from the center and ramp
    // off the density based on this distance squared.
    diff.x = center.x - direction2.x;
    diff.z = center.z - direction2.z;
    dist_sq = diff.x*diff.x + diff.z*diff.z;
    if(dist_sq > rad_sq2)
        {*density=0; return;}
    *density = (1-dist_sq/rad_sq2 + fast_turb*.05)* parms[1];
    if(height > st_d_ramp)
        *density *= (1-ease( (height - st_d_ramp)/(d_ramp_length),
                             .5 , .5));
}
```

The statements that have been changed from the `smoke_stream` procedure are marked with three asterisks (***). As can be seen, the main changes are in calculating and using two helical paths based on the frame number. One path displaces the center of the cylinder, and the point being rendered is moved along the other path. After trials with only one helical path, it becomes clear that two helical paths give a better effect. Figure 8.6 shows the results of this `rising_smoke_stream` procedure. This figure contains three images from an animation of rising smoke.

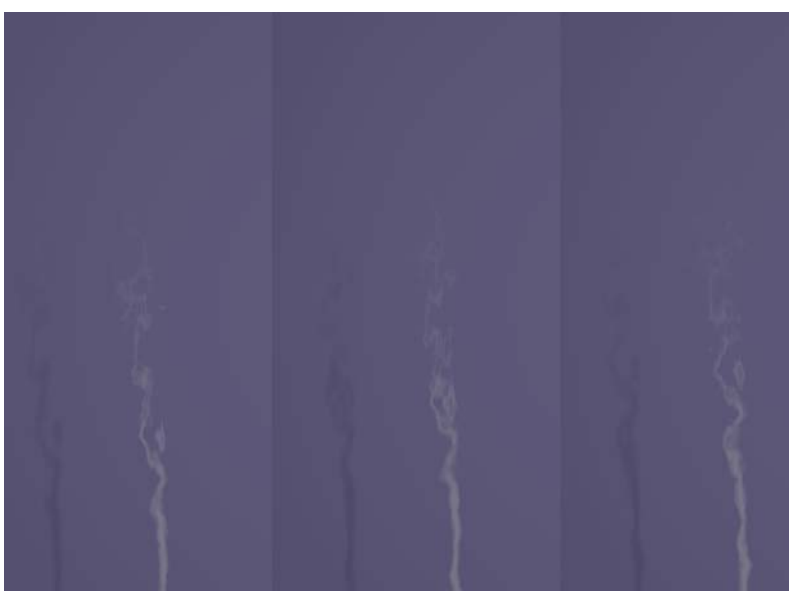


FIGURE 8.6 Rising column of smoke animation. Images are every 30 frames. Copyright © 1994 David S. Ebert.

Team LRN

THREE-DIMENSIONAL TABLES

As shown above, a wide variety of effects can be achieved through the use of helical paths. These aforementioned procedures require the same type of path to be used for movement throughout the entire volume of gas. Obviously, more complex motion can be achieved by having different path motions for different locations within the gas. A three-dimensional table specifying different procedures for different locations within the volume is a good, flexible solution for creating complex motion in this manner.

The use of three-dimensional tables (solid spaces) to control the animation of the gases is an extension of my previous use of solid spaces in which three-dimensional tables were used for volume shadowing effects (Ebert and Parent 1990).

The three-dimensional tables are handled in the following manner: The table surrounds the gas volume in world space, and values are stored at each of the lattice points in the table (see Figure 8.7). These values represent the calculated values for that specific location in the volume. To determine the values for other locations in the volume, the eight table entries forming the parallelepiped surrounding the point are interpolated. For speed in accessing the table values, I currently require table dimensions to be powers of two and actually store the three-dimensional table as a one-dimensional array. This restriction allows the use of simple bit-shifting operations in determining the array index. These tables could be extended to have nonuniform spacing between table entries within each dimension, creating an octree-like

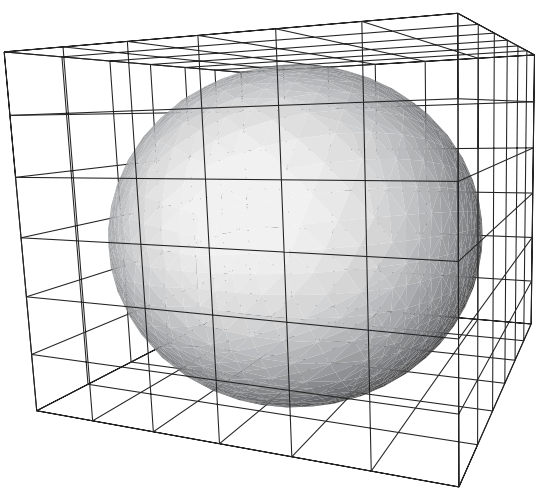


Figure 8.7 Three-dimensional table surrounding a sphere.

Team LRN

structure; however, this would greatly increase the time necessary to access values from the table, as this fast bit-shifting approach could no longer be used. Table dimensions are commonly of the order of 64 × 64 × 64 or 128 × 64 × 32.

I use two types of tables for controlling the motion of the gases: vector field tables and functional flow field tables. The vector field tables store direction vectors, density scaling factors, and other information for their use at each point in the lattice. Therefore, these tables are suited for visualizing computational fluid dynamics simulations or using external programs for controlling the gas motion. The vector field tables will not be described in this chapter. A thorough description of their use and merits can be found in Ebert (1991). This chapter concentrates on the use of functional flow field tables for animation control.

The functional flow field and vector field tables are incorporated into the volume density functions for controlling the shape and movement of the gas. Each volume density function has a default path and velocity for the gas movement. First, the default path and velocity are calculated; second, the vector field tables are evaluated; and, finally, functions that calculate direction vectors, density scaling factors, and so on, from the functional flow field tables are applied. The default path vector, the vector from the vector field table, and the vector from the flow field function are combined to produce the new path for the movement through the gas space.

Accessing the Table Entries

When values are accessed from these tables during rendering, the location of the sample point within the table is determined. As mentioned earlier, this point will lie within a parallelepiped formed by the eight table entries that surround the point. The values at these eight points are interpolated to determine the final value. The location within the table is determined by first mapping the three-dimensional screen space point back into world space. The following formula is then used to find the location of the point within the table:

```
ptable.x = (point.x-table_start.x) * table_inv_step.x
ptable.y = (point.y-table_start.y) * table_inv_step.y
ptable.z = (point.z-table_start.z) * table_inv_step.z
```

`Ptable` is the location of the point within the three-dimensional table, which is determined from `point`, the location of the point in world space. `table_start` is the location in world space of the starting table entry, and `table_inv_step` is the inverse of the step size between table elements in each dimension. Once the location within the table is determined, the values corresponding to the eight surrounding table entries are then interpolated (trilinear interpolation should suffice).

Team LRN

Functional Flow Field Tables

Functional flow field tables are a valuable tool for choreographing gas animation. These tables define, for each region of the gas, which functions to evaluate to control the gas movement. Each flow field table entry can contain either one specific function to evaluate or a list of functions to evaluate to determine the path for the gas motion (path through the gas space). For each function, a file is specified that contains the type of function and parameters for that function. The functions evaluated by the flow field tables return the following information:

- Direction vector
- Density scaling value
- Percentage of vector to use
- Velocity

The advantage of using flow field functions is that they can provide infinite detail in the motion of the gas. They are not stored at a fixed resolution, but are evaluated for each point that is volume rendered. The disadvantage is that the functions are much more expensive to evaluate than simply interpolating values from the vector field table.

The "percentage of vector to use" value in the previous list is used to provide a smooth transition between control of the gas movement by the flow field functions, the vector field tables, and the default path of the gas. This value is also used to allow a smooth transition between control of the gas by different flow field functions. This value will decrease as the distance from the center of control of a given flow field function increases.

Functional Flow Field Functions

Two powerful types of functions for controlling the movement of the gases are attractors/repulsors and vortex functions. Repulsors are the *exact* opposite of attractors, so only attractors will be described here. To create a repulsor from an attractor, simply negate the direction vector.

All of the following procedures will take as input the location of the point in the solid space (`pnt`) and a structure containing parameters for each instance of the function (`ff`). These procedures will return a density scaling factor (`density_scaling`), the direction vector for movement through the gas space (`direction`), the percentage of this vector to use in determining the motion through the gas space (`percent_to_use`), and a velocity scaling factor (`velocity`). The `density_scaling`

parameter allows these procedures to decrease or increase the gas density as it moves through a region of space. The `velocity` parameter similarly allows these procedures to change the velocity of the gas as it moves through a region of space. The most important parameters, however, are the `direction` and `percent_to_use` parameters, which are used to determine the path motion through the solid space.

Attractors

Attractors are primitive functions that can provide a wide range of effects. Figure 8.8 shows several frames of an attractor whose attraction increases in strength over time. Each attractor has a minimum and maximum attraction value. In this figure, the interpolation varies over time between the minimum and maximum attraction values of the attractor. By animating the location and strength of an attractor, many different effects can be achieved. Effects such as a breeze blowing (see Figure 8.9) and the wake of a moving object are easy to create. Spherical attractors create paths radially away from the center of attraction (as stated previously, path movement needs to be in the opposite direction of the desired visual effect). The following is an example of a simple spherical attractor function:

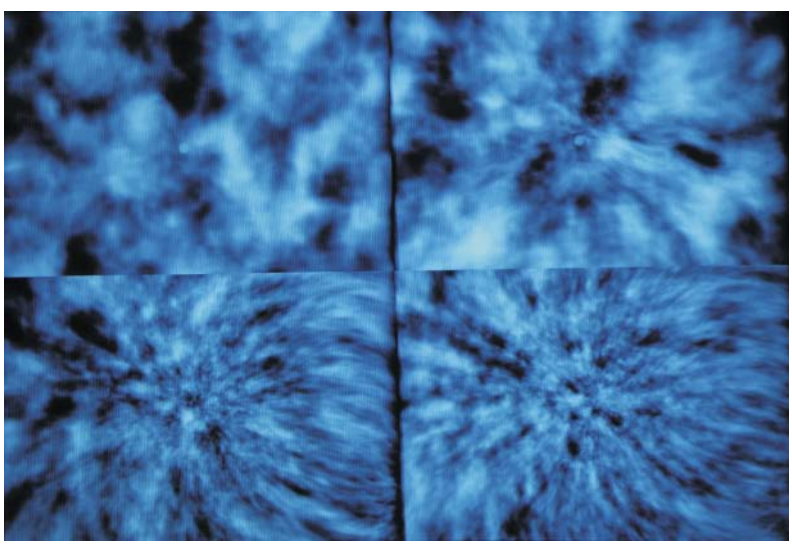


Figure 8.8 Effect of a spherical attractor increasing over time. Images are every 45 frames. The top-left image has 0 attraction. The lower-right image has the maximum attraction. Copyright © 1992 David S. Ebert.

Team LRN

FIGURE 8.9 An increasing breeze blowing toward the right created by an attractor. Copyright © 1991 David S. Ebert.

```
void spherical_attractor(xyz_td point, flow_func_td ff, xyz_td
                         *direction, float *density_scaling,
                         float *velocity, float *percent_to_use)
 {
 float dist, d2;
 // calculate distance & direction from center of attractor
 XYZ_SUB(*direction, point, ff.center);
 dist=sqrt(DOT_XYZ(*direction,*direction));
 // set the density scaling and the velocity to 1
 *density_scaling=1.0;
 *velocity=1.0;
 // calculate the falloff factor (cosine)
 if(dist > ff.distance)
    *percent_to_use=0;
 else if (dist < ff.falloff_start)
    *percent_to_use=1.0;
 else
   { d2 =(dist-ff.falloff_start)/(ff.distance-
                                  ff.falloff_start);
 *percent_to_use = (cos(d2*M_PI)+1.0)*.5;
   }
}
```

The `flow_func_td` structure contains parameters for each instance of the spherical attractor. The parameters include the center of the attractor (`ff.center`), the effective distance of attraction (`ff.distance`), and the location to begin the falloff from the attractor path to the default path (`ff.falloff_start`). This function ramps the use of the attractor path from `ff.falloff_start` to `ff.distance`. A cosine function is used for a smooth transition between the path defined by the attractor and the default path of the gas.

Extensions of Spherical Attractors

Variations on this simple spherical attractor include moving attractors, angle-limited attractors, attractors with variable maximum attraction, nonspherical attractors, and, of course, combinations of any or all of these types.

One variation on the preceding spherical attractor procedure is to animate the location of the center of attraction. This allows for dynamic animation control of the gas. Another useful variation is angle-limited attractors. As opposed to having the range of the attraction being 360 degrees, an axis and an angle for the range of attraction can be specified. This can be implemented in a manner very similar to angle-limited light sources and can be animated over time. These two variations can be combined to produce interesting effects. For example, an angle-limited attractor following the movement of the object can create a wake from a moving object. This attractor will cause the gas behind the object to be displaced and pulled in the direction of the moving object. The minimum and maximum attraction of the attractor can also be animated over time to produce nice effects as seen in Figures 8.8 and 8.9. Figure 8.8 shows an attractor increasing in strength over time, and Figure 8.9 shows a breeze blowing the steam rising from a teacup. As will be described later, the breeze is simulated with an animated attractor.

The geometry of the attraction can be not only spherical, but also planar or linear. A linear attractor can be used for creating the flow of a gas along a wall, as will be explained later.

Spiral Vortex Functions

Vortex functions have a variety of uses, from simulating actual physical vortices to creating interesting disturbances in flow patterns as an approximation of turbulent flow. The procedures described are not attempts at a physical simulation of vortices—an extremely complex procedure requiring large amounts of supercomputer time for approximation models.

Team LRN

One vortex function is based on the simple 2D polar coordinate function

$$r = \theta$$

which translates into three-dimensional coordinates as

$$x = \theta \times \cos(\theta)$$
$$y = \theta \times \sin(\theta)$$

The third dimension is normally linear movement over time along the third axis. To animate this function, θ is based on the frame number. To increase the vortex action, a scalar multiplier for the sine and cosine terms based on the distance from the vortex's axis is added. This polar equation alone produces swirling motion; however, more convincing vortices can be created by the modifications described below, which base the angle of rotation on both the frame number and the distance from the center of the vortex. The resulting vortex procedure is the following:

```
void calc_vortex(xyz_td *pt, flow_func_td *ff, xyz_td *direction,
    float *velocity, float *percent_to_use, int frame_num)
 {
 static tran_mat_td mat={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
 xyz_td     dir, pt2, diff;
 float      theta, dist, d2, dist2;
 float      cos_theta, sin_theta, compl_cos, ratio_mult;
 // calculate distance from center of vortex
XYZ_SUB(diff,(*pt), ff->center);
dist=sqrt(DOT_XYZ(diff,diff));
 dist2 = dist/ff->distance;
 // calculate angle of rotation about the axis
 theta = (ff->parms[0]*(1+.001*(frame_num)))/
         (pow((.1+dist2*.9), ff->parms[1]));
  // calculate matrix for rotating about the cylinder's axis
 calc_rot_mat(theta, ff->axis, mat);
transform_XYZ((long)1,mat,pt,&pt2);
 XYZ_SUB(dir,pt2,(*pt));
 direction->x = dir.x;
 direction->y = dir.y;
 direction->z = dir.z;
  // Have the maximum strength increase from frame parms[4] to
  // parms[5] to a maximum of parms[2]
 if(frame_num < ff->parms[4])
  ratio_mult=0;
 else if (frame_num <= ff->parms[5])
  ratio_mult = (frame_num - ff->parms[4])/
            (ff->parms[5] - ff->parms[4])* ff->parms[2];
```

```
else
 ratio_mult = ff->parms[2];
 //calculate the falloff factor
if(dist > ff->distance)
  { *percent_to_use=0;
   *velocity=1;
  }
else if (dist < ff->falloff_start)
  { *percent_to_use=1.0 *ratio_mult;
   // calc velocity
   *velocity= 1.0+(1.0 - (dist/ff->falloff_start));
  }
else
  { d2 =(dist-ff->falloff_start)/(ff->distance -
                                  ff->falloff_start);
   *percent_to_use = (cos(d2*M_PI)+1.0)*.5*ratio_mult;
   *velocity= 1.0+(1.0 - (dist/ff->falloff_start));
   }
}
```

This procedure uses the earlier polar function in combination with suggestions from Karl Sims (1990) to produce the vortex motion. For these vortices, both the frame number and the relative distance of the point from the center (or axis) of rotation determine the angle of rotation about the axis. The direction vector is then the vector difference of the transformed point and the original point. The `calc_vortex` procedure also allows the animation of the strength of the vortex action.

A third type of vortex function is based on the conservation of angular momentum: $r * q = constant$, where r is the distance from the center of the vortex. This formula can be used in the earlier vortex procedure to calculate the angle of rotation about the axis of the vortex: $\theta = (time * constant)/r$. The angular momentum will be conserved, producing more realistic motion.

An example of the effects achievable by the previous vortex procedure can be seen in Figure 8.10. Animating the location of these vortices produces interesting effects, especially when coordinating their movement with the movement of objects in the scene, such as a swirling wake created by an object moving through the gas.

Combinations of Functions

The real power of flow field functions is the ability to combine these primitive functions to control the gas movement through different volumes of space. The combination of flow field functions provides very interesting and complex gas motion. Two examples of the combination of flow field functions, wind blowing and flow into a hole, are presented next to illustrate the power of this technique.

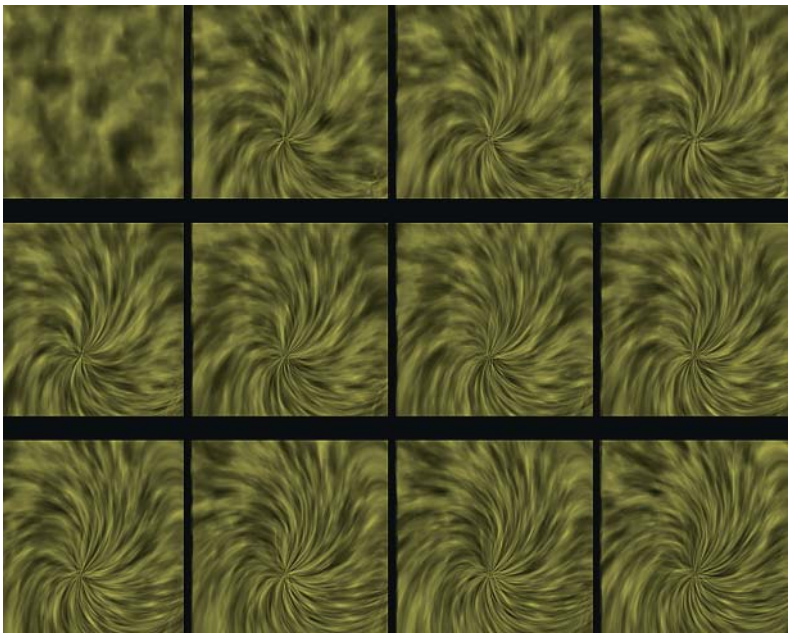


FIGURE 8.10 Spiral vortex. Images are every 21 frames. The top-left image is the default motion of the gas. The remaining images show the effects of the spiral vortex.
Copyright © 1994 David S. Ebert.

Wind Effects

The first complex gas motion example is wind blowing the steam rising from a teacup. A spherical attractor is used to create the wind effect. Figure 8.9 shows frames of an animation of a breeze blowing the steam from the left of the image. To produce this effect, an attractor was placed to the upper right of the teacup and the strength of attraction was increased over time. The maximum attraction was only 30%, producing a light breeze. An increase in the maximum attraction would simulate an increase in the strength of the wind. The top-left image shows the steam rising vertically with no effect of the wind. The sequence of images (top-right image to bottom-right image) shows the effect on the steam as the breeze starts blowing toward the right of the image. This is a simple combination of helical motion with an attractor. Notice how the volume of the steam, as well as the motion of the individual plumes, is "blown" toward the upper right. This effect was created by moving the center of the volume point for the ramping of the density over time. The x-value of the center

point is increased, based on the height from the cup and the frame number. By changing the `spherical_attractor` flow function and the `steam_moving` procedure given earlier, the blowing effect can be implemented. The following is the addition needed to the `spherical_attractor` procedure:

```
// ************************************************
// Move the Volume of the Steam
// Shifting is based on the height above the cup
// (parms[6]->parms[7]) and the frame range for increasing
// the strength of the attractor. This is from ratio_mult
// that is calculated above in calc_vortex.
// ************************************************
// Have the maximum strength increase from frame parms[4] to
// parms[5] to a maximum of parms[2]
if(frame_num < ff->parms[4])
  ratio_mult=0;
else if (frame_num <= ff->parms[5])
  ratio_mult = (frame_num - ff->parms[4])/
   (ff->parms[5] - ff->parms[4]) * ff->parms[2];
if(point.y < ff->parms[6])
   x_disp=0;
 else
 {if(point.y <= ff->parms[7])
   d2=COS_ERP((point.y-ff->parms[6])/
              (ff->parms[7]-ff->parms[6]));
  else
   d2=0;
   x_disp=(1-d2)*ratio_mult*parms[8]+fast_noise(point)*
          ff->parms[9];
}
return(x_disp);
```

Table 8.2 clarifies the use of all the parameters. The `ratio_mult` value for increasing the strength of the attraction is calculated in the same way as in the `calc_vortex` procedure. The `x_disp` value needs to be returned to the `steam_rising` function. This value is then added to the `center` variable before the density is ramped off. The following addition to the `steam_rising` procedure will accomplish this:

```
center = vol.shape.center;
center.x += x_disp;
```

Flow into a Hole in a Wall

The next example of combining flow field functions constrains the flow into an opening in a wall. The resulting images are shown in Figure 8.11. Figure 8.11(a)

TABLE 8.2 PARAMETERS FOR WIND EFFECTS

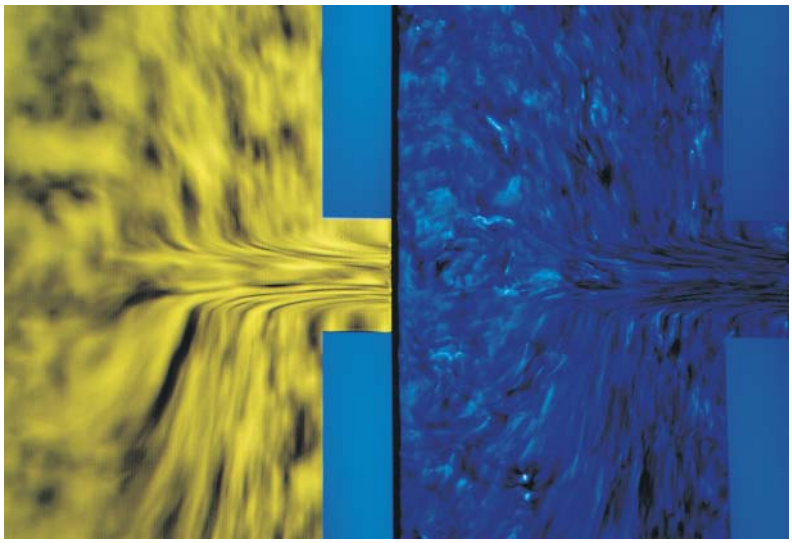
VARIABLE	DESCRIPTION
`point`	location of the point in world space
`ff -> parms[2]`	maximum strength of attraction
`ff -> parms[4]`	starting frame for attraction increasing
`ff -> parms[5]`	ending strength for attraction increasing
`ff -> parms[6]`	minimum *y*-value for steam displacement
`ff -> parms[7]`	maximum *y*-value for steam displacement
`ff -> parms[8]`	maximum amount of steam displacement
`ff -> parms[9]`	amount of noise to add in

shows gas flowing into an opening in a wall on the right of the image. Figure 8.11(b) shows liquid flowing into the opening. For this example, three types of functions are used. The first function is an angle-limited spherical attractor placed at the center of the hole. This attractor has a range of 180 degrees from the axis of the hole toward the left. The next function is an angle-limited repulsor placed at the same location, again with a range of repulsion of 180 degrees, but to the right of the hole. These two functions create the flow into the hole and through the hole. The final type of function creates the tangential flow along the walls. This function can be considered a linear attraction field on the left side of the hole. The line in this case would be through the hole and perpendicular to the wall (horizontal). This attractor has maximum attraction near the wall, with the attraction decreasing as the distance from the wall increases. As can be seen from the flow patterns toward the hole and along the wall in Figure 8.11, the effect is very convincing. This figure also shows how these techniques can be applied to hypertextures. Figure 8.11(b) is rendered as a hypertexture to simulate a (compressible) liquid flowing into the opening.

ANIMATING HYPERTEXTURES

All of the animation techniques described above can be applied to hypertextures; only the rendering algorithm needs to be changed. The volume density functions that I use for gases are, in reality, hypertexture functions. The difference is that an atmospheric rendering model is used. Therefore, by using a nongaseous model for illumination and for converting densities to opacities, the techniques described above will

Team LRN



(a) (b)

FIGURE 8.11 (a) Gas flowing into a hole in the wall. (b) Liquid flowing into a hole in the wall. Copyright © 1991 David S. Ebert.

produce hypertexture animations. An example of this is Figure 8.11(b). The geometry and motion procedures are the same for both of the images in Figure 8.11.

Volumetric Marble Formation

One other example of hypertexture animation will be explored: simulating marble formation. The addition of hypertexture animation to the solid texture animation discussed earlier will increase the realism of the animation considerably.

One approach is to base the density changes on the color of the marble. Initially, no turbulence will be added to the "fluid": density values will be determined in a manner similar to the marble color values, giving the different bands different densities. Just as in the earlier `marble_forming` procedure, turbulence will be added over time. In the following procedure, these changes are achieved by returning the amount of turbulence, `turb_amount`, from the solid texture function, `marble_forming`, described earlier. The density is based on the turbulence amount from the solid texture function. This is then shaped using the power function in a similar manner to the gas functions given before. Finally, a trick by Perlin (subtracting 0.5, multiplying

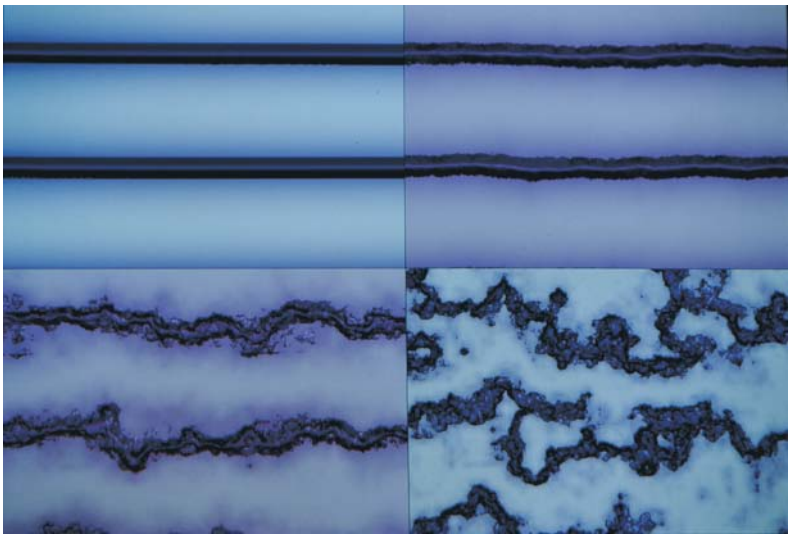


FIGURE 8.12 Liquid marble forming. Copyright © 1993 David S. Ebert.

by a scalar, adding 0.5, and limiting the result to the range of 0.2 to 1.0) is used to form a hard surface more quickly (Perlin 1992). The result of this function can be seen in Figure 8.12.

```
    //
    // parms[1] = maximum density value: density scaling factor
    // parms[2] = exponent for density scaling
    // parms[3] = x resolution for Perlin's trick (0-640)
    // parms[8] = 1/radius of fuzzy area for Perlin's trick(> 1.0)
    //
void molten_marble(xyz_td pnt, float *density, float *parms,
                   vol_td vol)
{
 float parms_scalar, turb_amount;
 turb_amount = solid_txt(pnt,vol);
 *density = (pow(turb_amount, parms[2]) )*0.35 +0.65;
 // Introduce harder surface more quickly.
 // parms[3] multiplied by 1/640
 *density *=parms[1];
 parms_scalar = (parms[3]*.0015625)*parms[8];
 *density= (*density-0.5)*parms_scalar +0.5;
 *density = MAX(0.2, MIN(1.0,*density));
}
```

PARTICLE SYSTEMS: ANOTHER PROCEDURAL ANIMATION TECHNIQUE

As previously mentioned, particle systems are different from the rest of the procedural techniques presented in this book in that their abstraction is in control of the animation and specification of the object. Particle systems were first used in computer graphics by Reeves (1983) to model a wall of fire for the movie *Star Trek II: The Wrath of Khan*. Since particle systems are a volumetric modeling technique, they are most commonly used to represent volumetric natural phenomena such as fire, water, clouds, snow, and rain (Reeves 1983). *Structured particle systems,* an extension of particle systems, have also been used to model grass and trees (Reeves and Blau 1985).

A particle system is defined by both a collection of geometric particles and the algorithms that govern their creation, movement, and death. Each geometric particle has several attributes, including its initial position, velocity, size, color, transparency, shape, and lifetime.

To create an animation of a particle system object, the following are performed at each time step (Reeves 1983):

- New particles are generated and assigned their attributes.
- Particles that have existed in the system past their lifetime are removed.
- Each remaining particle is moved and transformed by the particle system algorithms as prescribed by their individual attributes.
- These particles are rendered, using special-purpose rendering algorithms, to produce an image of the particle system.

The creation, death, and movement of particles are controlled by stochastic procedures, allowing complex, realistic motion to be created with a few parameters. The creation procedure for particles is controlled by parameters defining either the mean number of particles created at each time step and its variance or the mean number of particles created per unit of screen area at each time step and its variance. These values can be varied over time as well. The actual number of particles created is stochastically determined to be within *mean* + *variance* and *mean* − *variance*. The initial color, velocity, size, and transparency are also stochastically determined by mean and variance values. The initial shape of the particle system is defined by an origin, a region about this origin in which new generated particles are placed, angles

Team LRN

defining the orientation of the particle system, and the initial direction of movement for the particles.

The movement of particles is also controlled by stochastic procedures (stochastically determined velocity vectors). These procedures move the particles by adding their velocity vector to their position vector. Random variations can be added to the velocity vector at each frame, and acceleration procedures can be incorporated to simulate effects such as gravity, vorticity, conservation of momentum and energy, wind fields, air resistance, attraction, repulsion, turbulence fields, and vortices. The simulation of physically based forces allows realistic motion and complex dynamics to be displayed by the particle system, while being controlled by only a few parameters. Besides the movement of particles, their color and transparency can also change dynamically to give more complex effects. The death of particles is controlled very simply by removing particles from the system whose lifetimes have expired or who have strayed more than a given distance from the origin of the particle system.

The Genesis Demo sequence from *Star Trek II: The Wrath of Khan* is an example of the effects achievable by such a particle system. For this effect, a two-level particle system was used to create the wall of fire. The first-level particle system generated concentric, expanding rings of particle systems on the planet's surface. The second-level particle system generated particles at each of these locations, simulating explosions. During the Genesis Demo sequence, the number of particles in the system ranged from several thousand initially to over 750,000 near the end.

Reeves extended the use of particle systems to model fields of grass and forests of trees, calling this new technique structured particle systems (Reeves and Blau 1985). In structured particle systems, the particles are no longer an independent collection of particles, but rather form a connected, cohesive three-dimensional object and have many complex relationships among themselves. Each particle represents an element of a tree (e.g., branch, leaf) or part of a blade of grass. These particle systems are, therefore, similar to L-systems and graftals, specifically probabilistic, context-sensitive L-systems. Each particle is similar to a letter in an L-system alphabet, and the procedures governing the generation, movement, and death of particles are similar to the production rules. However, they differ from L-systems in several ways. First, the goal of structured particle systems is to model the visual appearance of whole collections of trees and grass, and not to correctly model the detailed geometry of each plant. Second, they are not concerned with biological correctness or modeling the growth of plants. Structured particle systems construct trees by recursively generating subbranches, with stochastic variations of parameters such as branching angle, thickness, and placement within a value range for each type of tree.

Team LRN

Additional stochastic procedures are used for placement of the trees on the terrain, random warping of branches, and bending of branches to simulate tropism. A forest of such trees can, therefore, be specified with a few parameters for distribution of tree species and several parameters defining the mean values and variances for tree height, width, first branch height, length, angle, and thickness of each species.

Both regular particle systems and structured particle systems pose special rendering problems because of the large number of primitives. Regular particle systems have been rendered simply as point light sources (or linear light sources for antialiased moving particles) for fire effects, accumulating the contribution of each particle into the frame buffer and compositing the particle system image with the surface-rendered image. No occlusion or interparticle illumination is considered. Structured particle systems are much more difficult to render, and specialized probabilistic rendering algorithms have been developed to render them (Reeves and Blau 1985). Illumination, shadowing, and hidden surface calculations need to be performed for the particles. Since stochastically varying objects are being modeled, approximately correct rendering will provide sufficient realism. Probabilistic and approximate techniques are used to determine the shadowing and illumination of each tree element. The particle's distance into the tree from the light source determines its amount of diffuse shading and probability of having specular highlights. Self-shadowing is simulated by exponentially decreasing the ambient illumination as the particle's distance within the tree increases. External shadowing is also probabilistically calculated to simulate the shadowing of one tree by another tree. For hidden surface calculations, an initial depth sort of all trees and a painter's algorithm are used. Within each tree, again, a painter's algorithm is used, along with a back-to-front bucket sort of all the particles. This will not correctly solve the hidden surface problem in all cases, but will give realistic, approximately correct images. Figure 8.13 contains images from the animation *The Adventures of André & Wally B,* illustrating the power of structured particle systems and probabilistic rendering techniques for structured particle systems.

Efficient rendering of particle systems is still an open active research problem (e.g., Etzmuss, Eberhardt, and Hauth 2000). Although particle systems allow complex scenes to be specified with only a few parameters, they sometimes require rather slow, specialized rendering algorithms. Simulation of fluids (Miller and Pearce 1989), cloth (Breen, House, and Wozny 1994; Baraff and Witkin 1998; Plath 2000), and surface modeling with oriented particle systems (Szeliski and Tonnesen 1992) are recent, promising extensions of particle systems. Sims (1990) demonstrated the suitability of highly parallel computing architectures to particle systems simulation.

Team LRN

FIGURE 8.13 Examples of probabilistic rendering of structure particle systems from *The Adventures of André and Wally B*. Courtesy of W. Reeves and R. Blau.

Team LRN

Particle systems, with their ease of specification and good dynamical control, have great potential when combined with other modeling techniques such as implicit surfaces (Witkin and Heckbert 1994) and volumetric procedural modeling.

Particle systems provide a very nice, powerful animation system for high-level control of complex dynamics and can be combined with many of the procedural techniques described in this book. For instance, turbulence functions are often combined with particle systems.

CONCLUSION

This chapter described several approaches to animating procedural models and showed several practical examples. The general animation approaches presented can be used with any procedural model or texture. The next chapter discusses how hybrid procedural models can be used to model and animate procedural clouds, while hardware acceleration techniques for these animation approaches can be found in Chapter 10.

Team LRN

9

VOLUMETRIC CLOUD MODELING WITH IMPLICIT FUNCTIONS

DAVID S. EBERT

Modeling clouds is a very difficult task because of their complex, amorphous structure and because even an untrained eye can judge the realism of a cloud model. Their ubiquitous nature makes them an important modeling and animation task. Several examples of the wide-ranging beauty of clouds can be seen in Figure 9.1. This chapter provides an introduction to the important visual and physical characteristics of clouds and important rendering issues for clouds. An overview of previous approaches to cloud modeling and animation, as well as recent approaches to interactive cloud modeling, is presented next. Finally, this chapter describes my volumetric procedural approach for cloud modeling and animation that allows easy, natural specification and animation of the clouds, provides the flexibility to include as much physics or art as desired into the model, unburdens the user from detailed geometry specification, and produces realistic volumetric cloud models. This technique combines the flexibility of volumetric procedural modeling with the smooth blending and ease of control of primitive-based implicit functions (meta-balls, blobs) to create a powerful new modeling technique. This technique also demonstrates the advantages of primitive-based implicit functions for modeling semitransparent volumetric objects. An exploration of how this cloud modeling approach can be adapted for real-time rendering can be found in Chapter 10.

CLOUD BASICS

Clouds are made of visible ice crystals and/or water droplets suspended in air, depending on the altitude of the cloud and, hence, the air temperature. Clouds are formed by air rising, cooling the water vapor in the air to its saturation point, and condensing onto small particles in the atmosphere. The visible condensed water vapor forms a cloud (University of Illinois 2002). Cloud shape varies based on the process that forces the air to rise or bubble up, the height at which the clouds form, and various other conditions (University of Illinois 2002). These air-lifting forces include

Team LRN

(a)

(b)

FIGURE 9.1 Several example photographs of real clouds. Copyright © 2002 David S. Ebert.

(c)

(d)

convection, convergence, lifting along frontal boundaries, lifting due to mountains (called *orographic lifting*), and Kelvin-Helmholtz shearing. Kelvin-Helmholtz shearing (billows) occurs when there is vertical wind shear and produces clouds that look similar to water waves or ripples. Several sources present a very nice introduction to clouds and their identification (University of Illinois 2002; Tricker 1970; Cotton and Anthes 1989; Houze 1993).

When considering cloud altitude, clouds formed above 20,000 feet (e.g., cirrus, cirrostratus) are often thin and wispy in appearance and composed primarily of ice crystals because of the cold temperatures. Clouds formed between 6500 feet and 23,000 feet (e.g., altocumulus) are primarily formed of water droplets and have the appearance of collections of small puffy clouds, sometimes in waves. Clouds formed below 6500 feet (e.g., stratus, stratocumulus) are, again, primarily comprised of water droplets and have the appearance of large clouds in layers. The most characteristic cloud type is the puffy cumulus cloud. Cumulus clouds are normally formed by convection or frontal lifting and can vary from having little vertical height to forming huge vertical towers (cumulonimbus) created by very strong convection.

The visual appearance of clouds not only is useful for creating more natural images and animations but is also very important for weather forecasters. Weather spotters are trained to look for key components of the visual appearance of clouds, enabling them to determine information on potential storms that cannot be collected from weather radar and other measurements. Clouds have several easily identifiable visual characteristics that must be modeled to produce accurate images and animations. First of all, clouds have a volumetrically varying amorphous structure with detail at many different scales. Second, cloud formation often results from swirling, bubbling, turbulent processes that produce the characteristic cloud patterns and their evolution over time. Finally, they have several illumination/shading characteristics that must be accurately rendered to obtain convincing images. Clouds are a three-dimensional medium of small ice and water droplets that absorb, scatter, and reflect light.

Illumination models for clouds are classified as low-albedo and high-albedo models. A low-albedo reflectance model assumes that secondary scattering effects are negligible, while a high-albedo illumination model calculates the secondary and higher-order scattering effects. For optically thick clouds, such as cumulus, stratus, and cumulonimbus, secondary scattering effects are significant, and high-albedo illumination models (e.g., Blinn 1982a; Kajiya and Von Herzen 1984; Rushmeier and Torrance 1987; Max 1994; Nishita, Nakamae, and Dobashi 1996) should be used. Detailed descriptions of implementing a low-albedo illumination algorithm can be found in several sources (Kajiya and Von Herzen 1984; Ebert and Parent 1990).

Team LRN

Simulation of wavelength-dependent scattering is also important to create correct atmospheric dispersion effects for sunrise and sunset scenes (see Figures 9.4 and 9.5 for example renderings of clouds with sunset illumination). Self-shadowing of clouds and cloud shadowing on landscapes are also important to create realistic images of cloud scenes and landscapes. Correct cloud shadowing requires volumetric shadowing techniques to create accurate images, which can be very expensive when volumetric ray tracing is used. As mentioned in Chapter 7, a much faster alternative is to use volumetric shadow tables (Kajiya and Von Herzen 1984; Ebert and Parent 1990) or hardware-based 3D texture slicing (Kniss, Kindlmann, and Hansen 2002).

SURFACE-BASED CLOUD MODELING APPROACHES

Modeling clouds in computer graphics has been a challenge for nearly 20 years (Dungan 1979), and major advances in cloud modeling still warrant presentation in the SIGGRAPH Papers Program (e.g., Dobashi et al. 2000). Many previous approaches have used semitransparent surfaces to produce convincing images of clouds. Voss (1983) introduced the idea of using fractal synthesis of parallel plane models to produce images of clouds seen from a distance. Gardner (1984, 1985, 1990) produces very impressive images and animations of clouds by using Fourier synthesis[1] to control the transparency of large, hollow ellipsoids. In his approach, large collections of ellipsoids define the general shape of the clouds, while the Fourier synthesis procedurally generates the transparency texture and creates the cloud detail. Kluyskens (2002) uses a similar approach to produce clouds in Alias/Wavefront's Maya animation system. He uses randomized, overlapping spheres to define the general cloud shape. A solid-texture "cloud texture" is then used to color the cloud and to control the transparency of the spheres. Finally, the transparency of the spheres near their edges is increased so that the defining spherical shape is not noticeable. Gardner's approach has been extended to real-time cloud rendering by using OpenGL extensions and adapting the transparency calculations for easier hardware implementation using subtracting blending modes (Elinas and Stürzlinger 2000).

Even simpler approaches to clouds are often used for interactive applications, such as games. One of the most common techniques is to create a 2D cloud texture that is texture mapped onto the sky dome of the scene. These textures are commonly generated using noise-based textures, and a real-time approach to generating procedural cloud textures using multitexturing and hardware blending can be found in

1. In this case, sums of cosines with different amplitudes and periods control the textured transparency.

Team LRN

Pallister (2002). The use of billboards and imposters[2] allows more interaction with the clouds in the scene. As with most other approaches using polygonal surfaces to approximate clouds, a Gaussian or exponential falloff of the transparency toward the edge of the cloud is needed so that the defining shape cannot be seen.

VOLUMETRIC CLOUD MODELS

Although surface-based techniques can produce realistic images of clouds viewed from a distance, these cloud models are hollow and do not allow the user to seamlessly enter, travel through, and inspect the interior of the cloud model. To capture the three-dimensional structure of a cloud, volumetric density-based models must be used. Kajiya and Von Herzen (1984) produced the first volumetric cloud model in computer graphics, but the results are not photorealistic. Stam and Fiume (1995) and Foster and Metaxas (1997) have produced convincing volumetric models of smoke and steam, but have not done substantial work on modeling clouds.

Neyret (1997) has recently produced some preliminary results of a convective cloud model based on general physical characteristics, such as bubbling and convection processes. This model may be promising for simulating convective clouds; however, it currently uses surfaces (large particles) to model the cloud structure. Extending this approach to volumetric cloud modeling and animation should produce very convincing images and animations.

Several researchers have combined volume modeling of clouds with surface-based imposter rendering. Mark Harris has recently extended the imposter technique to generate real-time flythroughs of complex, static cloudscapes (Harris and Lastra 2001; Harris 2002). Spherical particles are used to generate volumetric density distributions for clouds (approximately 200 particles per cloud), and a multiple scattering illumination model is precomputed for quicker shading of the particles. The particles are then rendered using splatting (Westover 1990) to create the imposters for the clouds. To allow the viewer to correctly fly through the environment and the actual clouds, the imposters need to be updated frequently. Dobashi et al. (2000) also have used imposters to allow quick rendering of clouds that are modeled using a partially physics-based cellular automata approach. However, the preintegration of the imposters for rendering limits the performance of this approach. As previously

2. An *imposter* is a texture-mapped polygon with the precomputed rendering of an object (color and alpha) mapped onto simple geometry to speed rendering. The texture-mapped image needs to be updated as the view changes to reflect the appropriate image of viewing the object from a new viewpoint.

Team LRN

mentioned, both of these approaches are actually volumetric cloud models, but use texture-mapped imposters for speed in rendering.

Particle systems (Reeves 1983) are commonly used to simulate volumetric gases, such as smoke, with very convincing results and provide easy animation control. The difficulty with using particle systems for cloud modeling is the massive number of particles that are necessary to simulate realistic clouds.

Several authors have used the idea of volume-rendered implicit functions for volumetric cloud modeling (Bloomenthal et al. 1997). Nishita has used volume-rendered implicits as a basic cloud model in his work on multiple scattering illumination models; however, this work has concentrated on illumination effects and not on realistic modeling of the cloud geometry (Nishita, Nakamae, and Dobashi 1996). Stam has also used volumetric blobbies to create his models of smoke and clouds (Stam and Fiume 1991, 1993, 1995). I have used volumetric implicits combined with particle systems and procedural detail to simulate the formation and geometry of volumetric clouds (Ebert 1997; Ebert and Bedwell 1998). This approach uses implicits to provide a natural way of specifying and animating the global structure of the cloud, while using more traditional procedural techniques to model the detailed structure. The implicits are controlled by a modified particle system that incorporates simple simulations of cloud formation dynamics, as described later in this chapter.

A VOLUMETRIC CLOUD MODELING SYSTEM

In developing this new cloud modeling and animation system, I have chosen to build upon the recent work in advanced modeling techniques and volumetric procedural modeling. As mentioned in Chapter 1, many advanced geometric modeling techniques, such as fractals (Peitgen, Jürgens, and Saupe 1992), implicit surfaces (Blinn 1982b; Wyvill, McPheeters, and Wyvill 1986; Nishimura et al. 1985), grammar-based modeling (Smith 1984; Prusinkiewicz and Lindenmayer 1990), and volumetric procedural models/hypertextures (Perlin 1985; Ebert, Musgrave, et al. 1994) use procedural abstraction of detail to allow the designer to control and animate objects at a high level. Their inherent procedural nature provides flexibility, data amplification, abstraction of detail, and ease of parametric control. When modeling complex volumetric phenomena, such as clouds, this abstraction of detail and data amplification are necessary to make the modeling and animation tractable. It would be impractical for an animator to specify and control the detailed three-dimensional density of a cloud model. This system does not use a physics-based approach

Team LRN

because it is computationally prohibitive and nonintuitive to use for many animators and modelers. Setting and animating correct physics parameters for dew point, particulate distributions, temperature and pressure gradients, and so forth is a time-consuming, detailed task. This model was developed to allow the modeler and animator to work at a much higher level and doesn't limit the animator by the laws of physics.

Volumetric procedural models have all of the advantages of procedural techniques and are a natural choice for cloud modeling because they are the most flexible, advanced modeling technique. Since a procedure is evaluated to determine the object's density, any advanced modeling technique, simple physics simulation, mathematical function, or artistic algorithm can be included in the model.

The volumetric cloud model uses a two-level model: the cloud macrostructure and the cloud microstructure. Implicit functions and turbulent volume densities model these, respectively. The basic structure of the cloud model combines these two components to determine the final density of the cloud.

Procedural `turbulence` and `noise` functions create the cloud's microstructure, in a manner similar to the `basic_gas` function (see Chapter 7). This allows the procedural simulation of natural detail to the level needed. Simple mathematical functions are added to allow shaping of the density distributions and control over the sharpness of the density falloff.

Implicit functions were chosen to model the cloud macrostructure because of their ease of specification and smoothly blending density distributions. The user simply specifies the location, type, and weight of the implicit primitives to create the overall cloud shape. Any implicit primitive, including spheres, cylinders, ellipsoids, and skeletal implicits, can be used to model the cloud macrostructure. Since these are volume rendered as a semitransparent medium, the whole volumetric field function is being rendered. In contrast, implicit surface modeling only uses a small range of values of the field to create the objects. The implicit density functions are primitive-based density functions: they are defined by summed, weighted, parameterized, primitive implicit surfaces. A simple example of the implicit formulation of a sphere centered at the point *center* with radius *r* is the following:

$$F(x,y,z):(x - \text{center}.x)^2 + (y - \text{center}.y)^2 + (z - \text{center}.z)^2 - r^2 = 0$$

The real power of implicit functions is the smooth blending of the density fields from separate primitive sources. I use Wyvill's standard cubic function (Wyvill, McPheeters, and Wyvill 1986) as the density (blending) function for the implicit primitives:

Team LRN

$$F_{\text{cub}}(r) = -\frac{4}{9}\frac{r^6}{R^6} + \frac{17}{9}\frac{r^4}{R^4} - \frac{22}{9}\frac{r^2}{R^2} + 1$$

In the preceding equation, r is the distance from the primitive. This density function is a cubic in the distance squared, and its value ranges from 1 when $r = 0$ (within the primitive) to 0 at $r = R$. This density function has several advantages. First, its value drops off quickly to zero (at the distance R), reducing the number of primitives that must be considered in creating the final surface. Second, it has zero derivatives at $r = 0$ and $r = R$ and is symmetrical about the contour value 0.5, providing for smooth blends between primitives. The final implicit density value is then the weighted sum of the density field values of each primitive:

$$\text{Density}_{\text{implicit}}(p) = \sum_i \left(w_i F_{\text{cub}_i}(p - q)\right)$$

where w_i is the weight of the ith primitive and q is the closest point on element i from p.

To create nonsolid implicit primitives, the location of the point is procedurally altered before the evaluation of the blending functions. This alteration can be the product of the procedure and the implicit function and/or a warping of the implicit space.

These techniques are combined into a simple cloud model:

```
volumetric_procedural_implicit_function(pnt, blend%, pixel_size)
  perturbed_point = procedurally alter pnt
                    using noise and turbulence
  density1 = implicit_function(perturbed_point)
  density2 = turbulence(pnt, pixel_size)
  blend = blend% * density1 +(1 - blend%) * density2
  density = shape resulting density based on user controls for
            wispiness and denseness (e.g., use pow and
            exponential function)
  return(density)
```

The density from the implicit primitives is combined with a pure turbulence-based density using a user-specified `blend%` (60% to 80% gives good results). The blending of the two densities allows the creation of clouds that range from entirely determined by the implicit function density to entirely determined by the procedural turbulence function. When the clouds are completely determined by the implicit

Team LRN

functions, they will tend to look more like cotton balls. The addition of the procedural alteration and turbulence is what gives them their naturalistic look.

VOLUMETRIC CLOUD RENDERING

This chapter focuses on the use of these techniques for modeling and animating realistic clouds. The volume rendering of the clouds is not discussed in detail. For a description of the volume-rendering system that was used to make my images of clouds in this book, please see Chapter 7. Any volume-rendering system can be used with these volumetric cloud procedures; however, to get realistic effects, the system should accumulate densities using atmospheric attenuation, and a physics-based illumination algorithm should be used. For accurate images of cumulus clouds, a high-albedo illumination algorithm (e.g., Max 1994; Nishita, Nakamae, and Dobashi 1996) is needed. Chapter 10 also shows how these techniques can be rendered at interactive rates using 3D texture mapping hardware.

Cumulus Cloud Models

Cumulus clouds are very common in nature and can be easily simulated using spherical or elliptical implicit primitives. Figure 9.2 shows the type of result that can be achieved by using nine implicit spheres to model a cumulus cloud. The animator/modeler simply positions the implicit spheres to produce the general cloud structure. Procedural modification then alters the density distribution to create the detailed wisps. The algorithm used to create the clouds in Figures 9.2 and 9.3 is the following:

```
void cumulus(xyz_td pnt, float *density, float *parms, xyz_td
             pnt_w, vol_td vol)
{
  float new_turbulence();     // my turbulence function
  float peachey_noise();      // Darwyn Peachey's noise function
  float metaball_evaluate();  // function for evaluating
                              // meta-ball primitives
  float mdens,                // meta-ball density value
       turb, turb_amount      // turbulence amount
       peach;                 // Peachey noise value
  xyz_td path;                // path for swirling the point
  extern int frame_num;
  static int ncalcd = 1;
  static float sin_theta_cloud, cos_theta_cloud, theta,
        path_x, path_y, path_z, scalar_x, scalar_y, scalar_z;
```

FIGURE 9.2 An example cumulus cloud. Copyright © 1997 David S. Ebert.

FIGURE 9.3 Example cloud creatures. Copyright © 1998 David S. Ebert.

```
  // calculate values that only depend on the frame number
  // once per frame
  if(ncalcd)
    {
     ncalcd = 0;
     // create gentle swirling in the cloud
    theta = (frame_num%600)*.01047196; // swirling effect
    cos_theta_cloud = cos(theta);
    sin_theta_cloud = sin(theta);
    path_x = sin_theta_cloud*.005*frame_num;
    path_y = .01215*(float)frame_num;
    path_z = sin_theta_cloud*.0035*frame_num;
    scalar_x = ( . 5+(float)frame_num*0.010);
    scalar_z = (float)frame_num*.0073;
    }
  // Add some noise to the point's location
  peach = peachey_noise(pnt); // Use Darwyn Peachey's noise
  pnt.x -= path_x - peach*scalar_x;
  pnt.y = pnt.y - path_y +.5*peach;
  pnt.z += path_z - peach*scalar_z;
  // Perturb the location of the point before evaluating the
  // implicit primitives.
  turb = fast_turbulence(pnt);
  turb_amount = parms[4]*turb;
  pnt_w.x += turb_amount;
  pnt_w.y -= turb_amount;
  pnt_w.z += turb_amount;
  mdens = (float)metaball_evaluate((double)pnt_w.x,
          (double)pnt_w.y, (double)pnt_w.z, (vol.metaball));
  *density = parms[1]*(parms[3]*mdens + (1.0 -
                       parms[3])*turb*mdens);
  *density = pow(*density,(double)parms[2]);
}
```

`Parms[3]` is the blending function value between implicit (meta-ball) density and the product of the turbulence density and the implicit density. This method of blending ensures that the entire cloud density is a product of the implicit field values, preventing cloud pieces from occurring outside the defining primitives. Using a large `parms[3]` generates clouds that are mainly defined by their implicit primitives and are, therefore, "smoother" and less turbulent. `Parms[1]` is a density scaling factor, `parms[2]` is the exponent for the `pow( )` function, and `parms[4]` controls the amount of turbulence to use in displacing the point before evaluation of the implicit primitives. For good images of cumulus clouds, useful values are the following: 0.2 < `parms[1]` < 0.4, `parms[2]` = 0.5, `parms[3]` = 0.4, and `parms[4]` = 0.7.

Team LRN

Cirrus and Stratus Clouds

Cirrus clouds differ greatly from cumulus clouds in their density, thickness, and fall-off. In general, cirrus clouds are thinner, less dense, and wispier. These effects can be created by altering the parameters to the `cumulus` cloud procedure and also by changing the implicit primitives. The density value parameter for a cirrus cloud is normally chosen as a smaller value and the exponent is chosen larger, producing larger areas of no clouds and a greater number of individual clouds. To create cirrus clouds, the user can simply specify the global shape (envelope) of the clouds with a few implicit primitives or specify implicit primitives to determine the location and shape of each cloud. In the former case, the shape of each cloud is mainly controlled by the volumetric procedural function and turbulence simulation, as opposed to cumulus clouds where the implicit functions are the main shape control. It is also useful to modulate the densities along the direction of the jet stream to produce more natural wisps. For instance, the user can specify a predominant direction of wind flow and use a turbulent version of this vector to control the densities as follows:

```
void Cirrus(xyz_td pnt, float *density, float *parms, xyz_td
           pnt_w, vol_td vol, xyz_td jet_stream)
{
 float new_turbulence();     // my turbulence function
 float peachey_noise();      // Darwyn Peachey's noise function
 float metaball_evaluate();  // function for evaluating the
                             // meta-ball primitives
 float mdens,                // meta-ball density value
    turb,turb_amount         // turbulence amount
    peach;                   // Peachey noise value
    xyz_td path;             // path for swirling the point
 extern int frame_num;
 static int ncalcd = 1;
 static float sin_theta_cloud, cos_theta_cloud, theta,
      path_x, path_y, path_z, scalar_x, scalar_y, scalar_z;
 // calculate values that only depend on the frame number
 // once per frame
if(ncalcd)
   {
     ncalcd = 0;
     //create gentle swirling in the cloud
     theta = (frame_num%600)*.01047196; // swirling effect
     cos_theta_cloud = cos(theta);
     sin_theta_cloud = sin(theta);
     path_x = sin_theta_cloud*.005*frame_num;
```

Team LRN

```
    path_y = .01215*(float)frame_num;
    path_z = sin_theta_cloud*.0035*frame_num;
    scalar_x = (.5+(float)frame_num*0.010);
    scalar_z = (float)frame_num*.0073;
  }
// Add some noise to the point's location
peach = peachey_noise(pnt); // Use Darwyn Peachey's noise
pnt.x -= path_x - peach*scalar_x;
pnt.y = pnt.y - path_y +.5*peach;
pnt.z += path_z - peach*scalar_z;
// Perturb the location of the point before evaluating the
// implicit primitives.
turb = fast_turbulence(pnt);
turb_amount = parms[4]*turb;
pnt_w.x += turb_amount;
pnt_w.y -= turb_amount;
pnt_w.z += turb_amount; // make the jet stream turbulent
jet_stream.x += .2*turb;
jet_stream.y += .3*turb;
jet_stream.z += .25*turb;
// warp point along the jet stream vector
pnt_w = warp(jet_stream, pnt_w);
mdens = (float)metaball_evaluate((double)pnt_w.x,
        (double)pnt_w.y, (double)pnt_w.z, (vol.metaball));
*density = parms[1]*(parms[3]*mdens + (1.0 - parms[3])*
                                      turb*mdens);
*density = pow(*density,(double)parms[2]);
  }
```

Examples of cirrus cloud formations created using these techniques can be seen in Figures 9.4 and 9.5. Figure 9.5 shows a higher cirrostratus layer created by a large elliptical primitive and a few individual lower cirrus clouds created with cylindrical primitives.

Stratus clouds can also be modeled by using a few implicits to create the global shape or extent of the stratus layer, while using volumetric procedural functions to define the detailed structure of all of the clouds within this layer. Stratus cloud layers are normally thicker and less wispy, as compared with cirrus clouds. Adjusting the size of the turbulent space (smaller/fewer wisps), using a smaller exponent value (creates more of a cloud layer effect), and increasing the density of the cloud can create stratus clouds. Using simple mathematical functions to shape the densities can create some of the more interesting stratus effects, such as a mackerel sky. The mackerel stratus cloud layer in Figure 9.6 was created by modulating the densities with turbulent sine waves in the x and y directions.

Team LRN

FIGURE 9.4 Cirrus clouds. Copyright © 1998 David S. Ebert.

FIGURE 9.5 Another example of cirrus and cirrostratus clouds. Copyright © 1998 David S. Ebert.

Team LRN

FIGURE 9.6 A mackerel stratus layer. Copyright © 1998 David S. Ebert.

Cloud Creatures

The combination of implicit functions with volumetric procedural models provides an easy-to-use system for creating realistic clouds, artistic clouds, and cloud creatures. Some examples of cloud creatures created using a simple graphical user interface (GUI) to position nine implicit spheres can be seen in Figure 9.3. They were designed in less than 15 minutes each, and a straw poll shows that viewers have seen many different objects in them (similar to real cloud shapes). Currently, the simple GUI only allows access to a small portion of the system. The rest of the controls are available through a text-based interface. Allowing the user access to more of the controls, implicit primitives, and parameters of the full cloud modeling system can create more complex shapes, time-based deformations, and animations. These cloud creatures are easily designed and animated by controlling the implicit primitives and procedural parameters. The implicit primitives blend and deform smoothly, allowing the specification and animation of skeletal structures, and provide an intuitive interface to modeling amorphous volumetric creatures.

User Specification and Control

Since the system uses implicit primitives for the cloud macrostructure, the user creates the general cloud structure by specifying the location, type, and weight of each implicit primitive. For the image in Figure 9.2, nine implicit spheres were positioned to create the cumulus cloud. Figure 9.3 shows the wide range of cloud shapes and

Team LRN

creatures that can be created by simply adjusting the location of each primitive and the overall density of the model through a simple GUI. The use of implicit primitives makes this a much more natural interface than with traditional procedural techniques. Each of the cloud models in this chapter was created in less than 30 minutes of design time.

The user of the system also specifies a density scaling factor, a power exponent for the density distribution (controls amount of wispiness), any warping procedures to apply to the cloud, and the name of the volumetric procedural function so that special effects can be programmed into the system.

ANIMATING VOLUMETRIC PROCEDURAL CLOUDS

The volumetric cloud models described earlier produce nice still images of clouds and also clouds that gently evolve over time. The models can be animated using the procedural animation techniques in Chapter 8 or by animating the implicit primitives. Procedural animation is the most flexible and powerful technique since any deformation, warp, or physical simulation can be added to the procedure. An animator using key frames or dynamics simulations can animate the implicit primitives. Several examples of applying these two animation techniques for various effects are described below.

Procedural Animation

Both the implicit primitives and the procedural cloud space can be animated algorithmically. One of the most useful forms of implicit primitive animation is warping. A time-varying warp function can be used to gradually warp the shape of the cloud over time to simulate the formation of clouds, their movement, and their deformation by wind and other forces. Cloud formations are usually altered based on the jet stream. To simulate this effect, all that is needed is to warp the primitives along a vector representing the jet stream. This can be done by warping the points before evaluating the implicit functions. The amount of warping can be controlled by the wind velocity or gradually added in over time to simulate the initial cloud development. Implicits can be warped along the jet stream as follows:

```
perturb_pnt = procedurally alter pnt using noise and turbulence
height = relative height of perturb_pnt
vector = jet_stream + turbulence(pnt)
perturb_pnt = warp(perturb_pnt, vector, height)
densityl = implicit_function(perturbed_pnt)
. . .
```

Team LRN

To get more natural effects, it is useful to alter each point by a small amount of turbulence before warping it. Several frames from an animation of a cumulus cloud warping along the jet stream can be seen in Figure 9.7. To create this effect, ease-in and ease-out based on the frame number were used to animate the warp amount. The implicit primitives' locations do not move in this animation, but the warping function animates the space to move and distort the cloud along the jet stream vector. Other warping functions to simulate squash and stretch (Bloomenthal et al. 1997) and other effects can also be used. Instead of a single vector and velocity, a vector field is input into the program to define more complex weather patterns. The current system allows the specification of vector flow tables and tables of functional primitives (attractors, vortices) to control the motion and deformation of the clouds. Stam (1995) used this procedural warping technique successfully in animating gases.

Implicit Primitive Animation

The implicit primitives can be animated in the same manner as implicit surfaces: each primitive's location, parameters (e.g., radii), and weight can be animated over time. This provides an easy-to-use, high-level animation interface for cloud animation. This technique was used in the animation "A Cloud Is Born" (Ebert et al. 1997), showing the birth of a cumulus cloud followed by a flythrough of it. Several stills from the formation sequence can be seen in Figure 9.8. For this animation, the centers of the implicit spheres were moved over time to simulate three separate cloud elements merging and growing into a full cumulus cloud. The radii of the spheres were also increased over time. Finally, to create animation in the detailed cloud

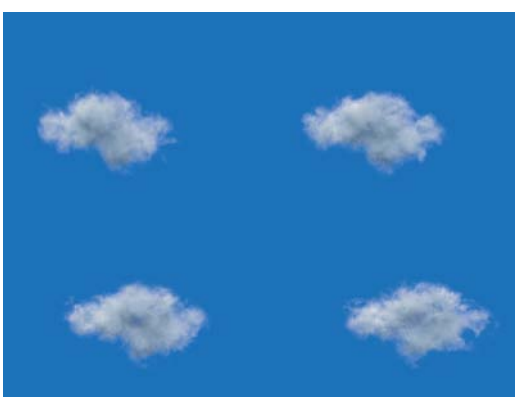


FIGURE 9.7 Cloud warping along the jet stream. Copyright © 1998 David S. Ebert.

Team LRN

FIGURE 9.8 Several stills from "A Cloud Is Born" showing the formation of the cloud. Copyright © 1998 David S. Ebert.

structure, each point was moved along a turbulent path over time before evaluation of the `turbulence` function, as illustrated in the `cumulus` procedure.

A powerful animation tool for volumetric procedural implicit functions is the use of dynamics and physics-based simulations to control the movement of the implicits and the deformation of space. Since the implicits are modeling the macrostructure of the cloud while procedural techniques are modeling the microstructure, fewer primitives are needed to achieve complex cloud models. Dynamics simulations can be applied to the clouds by using particle system techniques, with each particle representing a volumetric implicit primitive. The smooth blending and procedurally generated detail allow complex results with less than a few hundred primitives, a factor of 100–1000 less than needed with traditional particle systems. I have implemented a simple particle system for volumetric procedural implicit particles. The user specifies a few initial implicit primitives and dynamics information, such as speed, initial velocity, force function, and lifetime, and the system generates the location, number, size, and type of implicit for each frame. In our initial tests, it took less than one minute to generate and animate the implicit particles for 200 frames.

Team LRN

FIGURE 9.9 Volumetric procedural implicit particle system formation of a cumulus cloud.
Copyright © 1998 David S. Ebert.

Unlike traditional particle systems, cloud implicit particles never die—they just become dormant.

Cumulus clouds created through this volumetric procedural implicit particle system can be seen in Figure 9.9. The stills in Figure 9.9 show a cloud created by an upward turbulent force. The number of children created from a particle was also controlled by the turbulence of the particle's location. For the animations in this figure, the initial number of implicit primitives was 12, and the final number was approximately 50.

The animation and formation of cirrus and stratus clouds can also be controlled by the use of a volumetric procedural implicit particle system. For the formation of a large area of cirrus or cirrostratus clouds, the particle system can randomly seed space and then use turbulence to grow the clouds from the creation of new implicit primitives, as can be seen in Figure 9.10. The cirrostratus layer in this image contains 150 implicit primitives that were generated from the user specifying 5 seed primitives.

To control the dynamics of the cloud particle system, any commercial particle animation program can also be used. A useful approach for cloud dynamics is to use

Team LRN

FIGURE 9.10 Formation of a cirrostratus cloud layer using volumetric procedural implicit particles. Copyright © 1998 David S. Ebert.

qualitative dynamics—simple simulations of the observed properties and formation of clouds. The underlying physical forces that create a wide range of cloud formations are extremely complex to simulate, computationally expensive, and very restrictive. The incorporation of simple, parameterized rules that simulate observable cloud behavior will produce a powerful cloud animation system.

INTERACTIVITY AND CLOUDS

Creating models of clouds that can be rendered and animated at interactive rates is an exciting challenge, which is becoming tractable with the latest increases in graphics hardware speed and programmability.

Simple Interactive Cloud Models

There are several physical processes that govern the formation of clouds. Simple visual simulations of these techniques with particle systems can be used to produce more convincing cloud formation animations. As mentioned previously, Neyret

(1997) suggests that the following physical processes are important to cloud formation simulations: Rayleigh-Taylor instability, bubbles, temperature rate variation, Kelvin-Helmholtz instability, vortices, and Bernard cells. His model also takes into account phenomena at various scales, bubble generation, and cloud evolution.

Results from an implementation of these techniques by Ruchigartha using MEL scripts in Maya can be found at *www.ece.purdue.edu/~ebertd/ruchi1*. An example of the output from her system and the GUI cloud control can be seen in Figure 9.11.

Rendering Clouds in Commercial Packages

The main component needed to effectively render clouds is volume-rendering support in your renderer. Volumetric shadows, low- or high-albedo illumination, and correct atmospheric attenuation are needed to produce realistic-looking clouds. An example of a volume-rendering plug-in for Maya that can be used to create volume-rendered clouds can be found on the Maya Conductor CD from 1999 and on the HighEnd3D Web page, *www.highend3d.com/maya/plugins*. The plug-in, volumeGas, by Marlin Rowley and Vlad Korolov, implements a simplified version of my volume renderer (which was used to produce the images in these chapters). A resulting image from this plug-in can be seen in Figure 9.12.

CONCLUSION

The goal of these last three chapters has been to describe several techniques used to create realistic images and animations of gases and fluids, as well as provide insight into the development of these techniques. These chapters have shown a useful approach to modeling gases, as well as animation techniques for procedural modeling. To aid in reproducing and expanding the results presented here, all of the images are accompanied by detailed descriptions of the procedures used to create them. This gives you not only the opportunity to reproduce the results but also the challenge to expand upon the techniques presented. These chapters also provide insight into the procedural design approach that I use and will, hopefully, inspire you to explore and expand procedural modeling and animation techniques. The next chapter extends this work to discuss issues with real-time procedural techniques and hardware acceleration of these techniques.

Team LRN

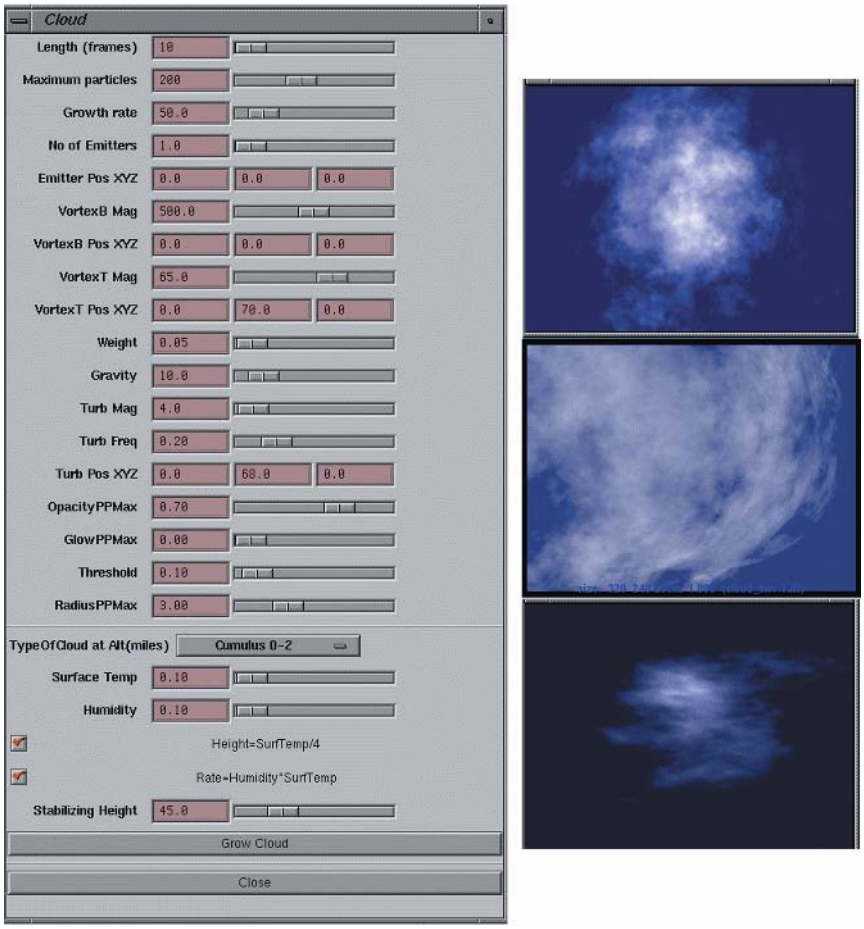


FIGURE 9.11 MEL GUI and sample images from Maya-based cloud particle system generator.

FIGURE 9.12. An example cloud rendering from a Maya plug-in. Image created by Marlin Rowley.

10

ISSUES AND STRATEGIES FOR HARDWARE ACCELERATION OF PROCEDURAL TECHNIQUES

DAVID S. EBERT

With contributions from Joe M. Kniss and Nikolai Svakhine

INTRODUCTION

As mentioned in Chapter 1, graphics hardware has reached a point in terms of speed and programmability that we can now start utilizing it for procedural texturing and modeling. This chapter will discuss general issues relating to hardware acceleration of procedural techniques, some common approaches to utilizing graphics hardware, and several examples showing how we can utilize the current generation of graphics hardware (as of 2002). With the rapid change in graphics technology, we should see more programmability and power for procedural techniques every year. One thing to remember in designing your application is that procedural techniques will continue to be a great solution to solve the data transfer bottleneck within your computer for years to come.

To clarify the presentation in the rest of the chapter, it is useful to review some basic terminology, which was introduced in Chapter 3. I'll refer to the graphics processing unit on the graphics board as the GPU. In most 2002-generation GPUs, there are two basic levels of processing: vertex processing and fragment processing. The vertex processing performs operations for each vertex (e.g., matrix and space transformations, lighting, etc.), and the fragment processing refers to processing that is performed per pixel-sized geometric primitive fragment. You are encouraged to read this chapter in conjunction with, or after, the discussion of real-time programmable shading by Bill Mark in Chapter 3. These chapters cover several similar issues, but each has a different perspective and different emphasis.

GENERAL ISSUES

In general, when migrating a procedural technique from CPU rendering to GPU rendering, there are several factors to consider, including the following:

Team LRN

- Language interface
- Precision
- Flexibility and capabilities
- Storage
- Levels of operation

The language interface is a basic issue that must be confronted, since most current standards don't provide all of the features of the graphics hardware at a high-level language (C, C++) interface. Several hardware manufacturers are currently developing better tools to simplify the task of programming the graphics hardware, but, currently, the only high-level language interface available is the Stanford real-time shading language (RTSL). This is a very basic issue that will, hopefully, be solved in the near term and adopted into the next generation of graphics standards (e.g., proposals for OpenGL 2.0).

The issue of computational precision will probably still exist for several years, and it must be considered when designing and implementing algorithms that will run on a variety of graphics platforms and game consoles. Therefore, when designing an algorithm, the precision at each stage in the pipeline must be considered. For instance, in many 2002-generation graphics cards and game consoles, vertex processing has higher precision than fragment processing, while the final texture combiners in the fragment-processing hardware have higher accuracy than earlier operations. These precision issues are most important for approximating detailed functions, when performing a large number of computations and iterative computations (e.g., multipass computations). Accumulation of error may have a significant effect on the results. Another issue that needs to be considered is the precision of the operands. For instance, if you are limited to 8-bit operands because the operands are coming from a texture map access, having 16-bit precision for an operation is not much of an advantage. Again, these precision issues have a cumulative effect and increase as the complexity of the algorithm increases. Finally, precision is also a factor in hardware interpolation and must be considered when using hardware blending and texture mapping interpolation to perform some computations.

Flexibility is a longer-term concern for hardware-accelerated procedural techniques. Most graphics hardware has limits on the operations that can be performed at each stage, and it may be necessary to reorder the operations in your software procedural technique to implement it using graphics hardware. For instance, with 2002-generation GPUs, you cannot create geometry on the fly within the GPU for

Team LRN

performance reasons, and vertex programs cannot access texture maps. The available operations are also a determining factor in designing GPU-accelerated procedural techniques. What mathematical functions are required? Are these available at the desired stage in the hardware pipeline? If not, can they be approximated using texture maps, tables, or registers? How many texture maps can be accessed in a single hardware pass? What is the performance trade-off between two-dimensional texture map accesses and three-dimensional texture map accesses?

The recent trend in the flexibility of graphics hardware operations is that, in general, vertex processing offers more program flexibility than fragment processing. The GPU program flexibility has a significant impact on the ease and effectiveness of real-time procedural texturing and modeling. As this flexibility increases, it will be possible to implement more complex and elegant procedural models and textures entirely within the GPU. Hopefully, within the next several years, we will be able to simply download an algorithm to the GPU and completely generate the model, as well as the textures, within the GPU.

As with hardware flexibility, available storage can significantly impact the design of procedural techniques. When decomposing an algorithm into vertex and fragment components, is there enough storage at each stage? The type of storage and the access penalty of the storage are important to consider. Having the ability to write to a texture map provides a large amount of storage for the algorithm; however, this will be much slower than writing and reading from a small set of registers. Access to texture map storage also allows information that doesn't change per frame to be precomputed and then accessed from the texture map, increasing the performance of the algorithm.

As can be seen from the above discussion, the various levels of operation in the graphics pipeline, in most cases, must be considered to develop efficient real-time procedural techniques. Some high-level language compilers (e.g., RTSL) can hide some of the architecture specifics from the user, but the understanding of at least the separation of vertex and fragment processing must be considered to develop effective interactive procedural techniques.

COMMON ACCELERATION TECHNIQUES

This section describes a number of common approaches to accelerate procedural techniques using graphics hardware. Many of these techniques are straightforward extensions of common software acceleration techniques and common hardware rendering acceleration techniques. Precomputation is probably the most common acceleration technique in graphics and can be used to some extent with procedural

Team LRN

techniques. However, the trade-off of performance from precomputation with the flexibility and detail provided by on-the-fly computation must be considered. Precomputing a marble texture and storing it in a 2D texture map would not be considered a procedural texture since the detail on demand and other benefits of procedural techniques would be lost. However, precomputing functional approximations (e.g., sine, cosine, limited power functions) for functions not available in the GPU program is commonly used to allow hardware implementation of procedural techniques.

Another common technique is the use of dummy geometry and depth culling. Most graphics hardware does not allow a vertex or fragment program to create new geometry. Therefore, in order to create a hardware procedural model, the common trick is to create a large list of dummy polygons that are initially created with a depth value that allows fast hardware culling of the geometry to reduce computations for unused dummy geometry. The algorithm that is creating "new" geometry simply transforms the geometry from the dummy list to make it visible and part of the procedural model. This can easily be done for a particle system that runs completely on the graphics hardware and could also be used to create L-system or grammar-based plant models that grow and evolve based on algorithms running on the graphics hardware.

A good algorithm designer can also harness the power of the components of the graphics hardware to perform tasks for which they weren't originally designed. Most GPUs have hardware that performs interpolation, dot products, blending, and filtering. By storing values appropriately into textures, registers, operands of different combiner stages, and so on, you can use the hardware to perform these operations as they are needed in your procedure. For instance, Pallister (2002) has used multitexturing, texture blending, and built-in hardware filtering and interpolation for quickly computing clouds for sky domes. The hardware dot product capability can also be used for space transformation to compute warps and procedures that use different texture/model spaces.

The use of multilevel procedural models can be easily adapted to hardware to achieve higher-resolution models at interactive rates. The most flexible way to implement this approach takes advantage of the dependent texture read facility in many GPUs. This feature is vital for many procedural techniques and allows texture coordinates for accessing a texture to be computed based, at least partially, on the results of a previous texture lookup result. To create the multilevel model, a coarse model is created from a low-resolution texture map, and a separate detail texture map is combined to produce higher-frequency details, as illustrated with the interactive procedural cloud example below. Detail and noise textures can also create more natural

Team LRN

and artistic effects by using them to adjust the texture coordinates for accessing another texture map. These texture coordinates can also be animated over time by varying the amount of the detail/noise texture that is used in generating the new texture lookup.

Animation of procedural techniques in real time is a challenge. Updating large texture maps per frame is not currently feasible because of memory bandwidth limits. However, animating texture coordinates and coefficients of procedures can be performed in real time. If this approach is used in the design of the procedural model/texture, it is very easy to create animated real-time effects.

EXAMPLE ACCELERATED/REAL-TIME PROCEDURAL TEXTURES AND MODELS

Several examples of real-time procedural techniques are presented below to illustrate how to take the procedural techniques described throughout this book and adapt them to interactive graphics hardware.

Noise and Turbulence

One of the most common procedural functions used is the noise function, so this is a natural place to start describing how to implement procedural techniques in graphics hardware. Several different approaches have been taken to implement hardware noise and turbulence. Hart (2001) has used the programmability of the graphics hardware to actually calculate noise and turbulence functions by making many passes through the graphics hardware. The more common approach is to use 2D or 3D texture mapping hardware to generate a lattice-based noise function. With a 2D texture map, the third texture coordinate is used as a scaling factor for accessing different components of the 2D texture map. A 3D texture map implementation is very straightforward for implementation: simply compute a 3D texture map of random numbers. If you then set the texture mapping mode to repeat the texture throughout space, the texture mapping hardware will do the interpolation within the lattice for free.

The main difficulty that arises is in implementing turbulence and more complex noise functions (e.g., gradient noise). A turbulence function can easily be implemented by making several texture lookups into the 3D noise texture, scaled and summed appropriately. However, there may be a penalty with this method, since it requires several texture accesses (e.g., four or five) for the turbulence functions alone. Depending on your hardware, this may significantly limit the number of

Team LRN

texture accesses available in a single pass. The good news is that all of the texture accesses are to the same texture, and this does not introduce the overhead of multiple 3D texture maps. Of course, multiple levels of turbulence values can be precomputed and the result stored in the three-dimensional texture, but the replication of the turbulence function throughout space without seams becomes trickier. Also, this introduces smooth interpolation of the turbulence values within the lattice cells and decreases the fine detail of the turbulence function that can be produced for the same-size three-dimensional table. For instance, with a 64^3 noise table and four octaves of noise values summed (four texture accesses), the resulting resolution of the turbulence function is much greater than using a 64^3 turbulence texture map.

Marble

A simple, classic procedural texture is the following marble function:

```
marble(pnt) = color_spline( sin(pnt.x + turbulence(pnt)) +1)*.5, white, blue)
```

With flexible dependent texture reads, this can be implemented in hardware using the following algorithm:

```
R1 = texture3D(turbulence, pnt) using 3D texture access (1 to n accesses)
R2 = sin_texture(pnt.x+R1) => offset dependent read
     1D texture contains (sin(value)+1)*.5
R3 = color_spline_texture(R2) => value dependent read
```

This implementation requires at least three texture accesses with two dependent texture reads. Additionally, the register combiners need to support the offset dependent read and replace dependent read operations.

An example implementation of this in NVIDIA's Cg language (version 1.01), with restrictions in vertex and fragment programmability based on the capabilities in the Nvidia GeForce3 graphics processor, follows. This illustrates how these techniques can be applied to programmable graphics hardware and also the importance of the flexibility of operations at the different levels of the graphics pipeline: restrictions in the dependent texture read operations make this example more complex than the previous example. Therefore, the marble formula is simplified and several values are precomputed into textures. This procedure allows the real-time change of the period of the sine function and the amount of turbulence/noise added. To make this example more robust and eliminate texture seams, the single 3D turbulence texture call should be replaced with several scaled 3D noise texture calls to implement the turbulence function in the graphics hardware. The procedure has a vertex program and a fragment program component that use two textures: T(s, t, r) and sine01(s, t).

Team LRN

The noise and coordinate texture, T(s, t, r), has elements of the following form:

T(s, t, r) =

Turb(s, t, r)	s, t, or r		
R	G	B	A

where

R-component – Turb(s, t, r) – turbulence at point (s, t, r).
G-component – s, t, or r, depending on desired marble orientation.
B, A – unused

The sine01(s, t) texture contains the values of the following function:

$$\text{sine01}(s, t) = (\sin((s+t)*2\text{pi})+1)*0.5$$

The color of the marble texture is computed to produce a blue-white marble using the following procedure:

```
VOID Sine(D3DXVECTOR4* pOut, D3DXVECTOR2* pTexCoord, D3DXVECTOR2* pTexelSize,
LPVOID pData)
{
         double sine;
         double R_, G_, B_;
         double t;
         sine = (sin((pTexCoord->x+pTexCoord->y)*2*MY_PI)+1)*0.5;

         if (sine < 0.2)
         {
           R_ = G_ = 255;
           B_ = 255;
         }
         else if (sine >= 0.2 && sine < 0.45)
         {
           t = (sine - 0.2)/0.25;
           B_ = 255;
           R_ = (1-t)*255;
           G_ = (1-t)*255;
         }
         else if (sine >= 0.45 && sine < 0.6)
         {
           R_ = G_ = 0;
           B_ = 255;
         }
         else if (sine >= 0.6 && sine < 0.8)
         {
           t = (sine - 0.6)/0.2;
```

Team LRN

```
            R_ = (t)*255;
            G_ = (t)*255;
            B_ = 255;
          }
          else if (sine >= 0.8)
          {
            R_ = G_ = 255;
            B_ = 255;
          }

  pOut->x = R_/255.0;
  pOut->y = G_/255.0;
  pOut->z = B_/255.0;
  pOut->w = 1.0;

return;

}
```

The following Cg vertex and marble shaders are based on NVIDIA's Direct3D implementation of DetailNormalMaps() that is part of the Cg Demo Suite.

Marble_Vertex.cg

```
struct a2v : application2vertex {
                                    float4 position;
                                    float3 normal;
                                    float2 texture;
                                    float3 S;
                                    float3 T;
                                    float3 SxT;
                                };

struct v2f : vertex2fragment {
                                    float4 HPOS;
                                    float4 COL0; //lightvector in tangent space
                                    float4 COL1; //normal in tangent space
                                    float4 TEX0;
                                    float4 TEX1;
                                    float4 TEX2;
                                };

v2f main(a2v I,
                                    uniform float4x4 obj2proj,
                                    uniform float3 lightVector, //in object space
                                    uniform float3 eyePosition, //in object space
                                    uniform float2 AB,
                                    uniform float4 offset,
```

```
                                    uniform float4 tile
                                    )
{
        v2f O;

/* Transform Position coordinates */

        // transform position to projection space

        float4 outPosition = mul(obj2proj, I.position);

        O.HPOS = outPosition;

/* Compute lightVector used for lighting */
        // compute the 3 x 3 transform from tangent space to object space
        float3x3 obj2tangent;
        obj2tangent[0] = I.S;
        obj2tangent[1] = I.T;
        obj2tangent[2] = I.SxT;

              // transform light vector from object space to tangent space and
              // pass it as a color
        O.COL0.xyz = 0.5 * mul(obj2tangent, lightVector) + 0.5.xxx;

              //Normal is (0, 0, 1) in tangent space
        O.COL1.xyz = float3(0, 0, 1);

/* Setting up noise and marble parameters */

        float A = AB.x; //A and B coefficients of marble formula, passed as
                        //uniform parameters to a shader
        float B = AB.y;

              //3D texture coordinate is just scaled and biased point coordinate
        O.TEX0 = tile.x*0.2*(I.position+offset);
        O.TEX0.w = 1.0f;

        //Those are the coordinates used for Sine calc in marble,
        //TEX1 and TEX2 are used as coordinates for dependent texture lookup
        //in pixel shader (Marble_Pixel.cg)

        O.TEX1.xyz = float3(A*B, 0, 0);
        O.TEX2.xyz = float3(0, A, 0);
      return O;
}
```

Team LRN

Marble_Pixel.cg

```
struct v2f : vertex2fragment {
              float4 HPOS;
              float4 COL0; //light vector in tangent space
              float4 COL1; //normal vector in tangent space
              float4 TEX0;
              float4 TEX1;
              float4 TEX2;
};

//fragout is standard connector fragment2framebuffer, outputting float4 color.

fragout main(v2f I,
             uniform sampler3D noiseTexture,
             uniform sampler2D sineTexture,
              )
{
             fragout O;
             //Texture coordinate interpolants, TEX0, TEX1, and TEX2 are set in
             //the vertex program (Marble_Vertex.cg)

             float4 noiseVal = tex3D(noiseTexture); //TEX0 is used here, getting
                                                    //turbulence/x value

             float4 marbleValue = tex2D_dp3x2(sineTexture, I.TEX1, noiseVal);

                  //TEX1 and TEX2 are used here.
                  //The coordinates for the sine lookup are
                  //S = A*B*Turb(x);
                  //T = A*P.x;
                  //So the result of lookup is sin(A*(P.x+B*Turb(x)).

             //Lighting part calculates parameter diffIntensity
             float ambient = 0.2f;
             float4 sNormal = (I.COL1);
             float4 lightVector = expand(I.COL0); //Calculated in vertex program
             float diffIntensity = uclamp(dot3(sNormal.xyz, lightVector.xyz))
               +ambient;

             //Final result
             O.col = marbleValue*diffIntensity;
             return O;
}
```

Results from this marble function, with varying periods of the sine function, can be seen in Figure 10.1.

Team LRN

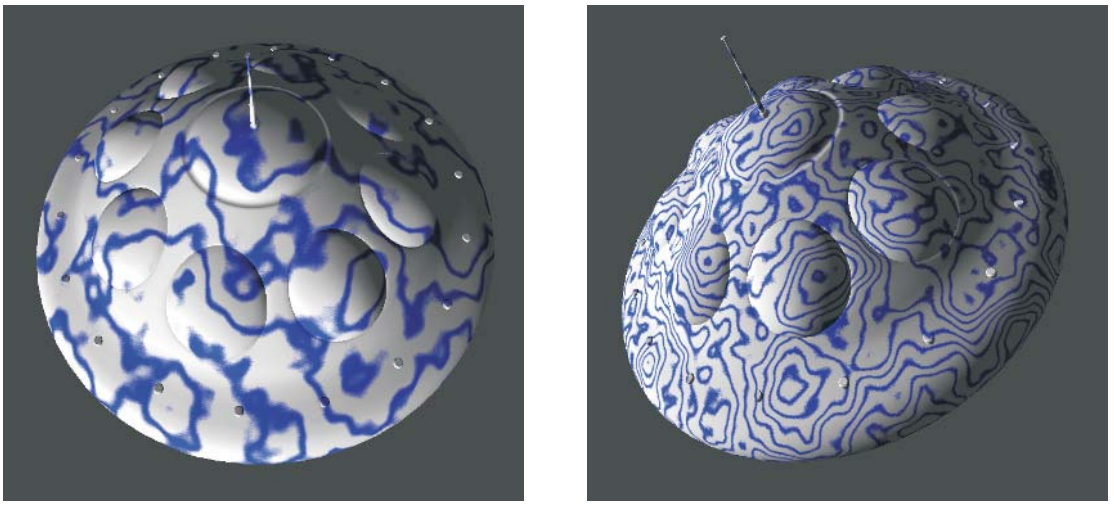


FIGURE 10.1 Example real-time marble textures created with varying periods of the sine function. Images created by Nikolai Svakhine.

Smoke and Fog

Three-dimensional space-filling smoke and fog can easily be created with 3D texture mapping hardware. The common approach is to implement volume rendering using a series of slicing planes that "slice" the volume and texture space. Evaluating shaders and 3D textures at each fragment in the slice plane samples the volume and performs the volume rendering. The shader should return the color and opacity of the given region in space. These slice plane polygons are then composited in a front-to-back or back-to-front order in the rendering process. For procedural smoke and fog, the actual opacity and color is determined by shaping a turbulence function generated from a 3D noise texture, as described in Chapter 7. A simple smoke function for determining opacity is the following:

$$\text{Smoke}(pnt) = (\text{turbulence}(pnt) * \text{scalar})^{\text{power}}$$

This can be implemented with three octaves of noise using the following procedure for each volume slice fragment:

```
  R1 = texture3D(noise, pnt)—1st octave of noise using 3D texture
pnt2 = pnt * {2.0,2.0,2.0,1}
```

```
  R2 = texture3D(noise, pnt2)—2nd octave of noise using 3D texture
  R2*= R1
pnt2 = pnt2 * {2.0,2.0,2.0,1}
  R3 = texture3D(noise, pnt2)—3rd octave of noise using 3D texture
  R3*= R2
  R4 = scalar—for general density scaling
```

or

```
  R4 = texture3D (scalar, pnt)—use a varying scalar texture for more
                               variation
  R5 = texture1D (power_texture, R3*R4)—only need 1D texture if power is
                                        constant
```

or

```
  R5 = texture2D(power_texture,R3*R4, power)—for multifractal effect where
                                       the power is varied throughout space
```

This requires three 3D texture accesses plus a dependent texture read. Depending on your hardware capabilities, the above general idea may need to be reorganized due to restrictions on the order of operations. Of course, a precomputed turbulence texture could be used, but a high-resolution 3D texture would be needed to get good results. To animate the function, a downward helical path is very effective in creating the impression of the smoke rising. Therefore, the point is first swirled downward before the calls to the noise texture. The swirling can be calculated using a simple time-dependent helical path (as described in Chapter 8). This helical path is generated by a cosine texture and the following formula, which adds another texture read and changes the noise texture read to a dependent texture offset read:

```
p2.x = r1*cos(theta)—theta varies with frame_number, r1 is one ellipse
       radii
p2.z = r2*cos(theta)—r2 is 2nd ellipse radii
p2.y = -const*frame_number
  R1 = texture3D(noise, pnt+p2)—using 3D dependent offset texture
...
```

Real-Time Clouds and Procedural Detail

High-frequency detail cannot be represented effectively using reasonably sized models that will fit in 3D texture memory on current-generation graphics boards. These high-frequency details are essential for capturing the characteristics of many volumetric objects such as clouds, smoke, trees, hair, and fur. Procedural noise simulation is a very powerful tool to use with small volumes to produce visually compelling

Team LRN

simulations of these types of volumetric objects. My approach for modeling these details is similar to my approach for modeling clouds, described in Chapter 9: use a coarse technique for modeling the macrostructure and use procedural noise-based simulations for the microstructure. These techniques can be adapted to interactive volume rendering through two volume perturbation approaches that are efficient on modern graphics hardware. The first approach is used to perturb optical properties in the shading stage, while the second approach is used to perturb the volume itself.

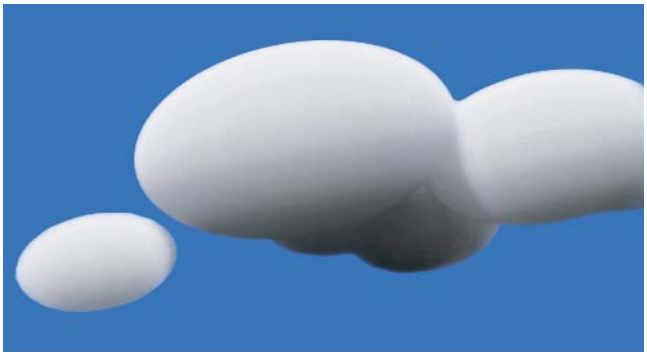
Both volume perturbation approaches employ a small 3D perturbation volume, 32^3. Each texel is initialized with four random 8-bit numbers, stored as RGBA components and blurred slightly to hide the artifacts caused by trilinear interpolation. Texel access is then set to repeat. Again, depending on the graphics hardware, an additional rendering pass may be needed for both approaches because of limitations imposed on the number of textures that can be simultaneously applied to a polygon and the number of sequential dependent texture reads permitted. The additional pass occurs before the shading volume-rendering pass.

With a volume object, multiple copies of the 3D noise texture are applied to each volume slice at different scales. They are then weighted and summed per pixel. To animate the perturbation, we add a different offset to each noise texture's coordinates and update it each frame.

The first approach uses my lattice-based noise described in Chapter 7 to modify the optical properties of the volume-rendered object using four per-pixel noise components. This approach makes the materials appear to have inhomogeneities. The user may select which optical properties are modified, and this technique can produce the subtle iridescent effects seen in Figure 10.2(c).

The second approach is closely related to Peachey's vector-based noise simulation described in Chapter 2. It uses the noise to modify the location of the data access for the volume. In this case, three components of the noise texture form a vector, which is added to the texture coordinates for the volume data per pixel. The data is then read using a dependent texture read. The perturbed data is rendered to a pixel buffer that is used instead of the original volume data. The perturbation texture is applied to the polygon twice, once to achieve low frequency with high-amplitude perturbations and again to achieve high frequency with low-amplitude perturbations. Allowing the texture to repeat creates the high-frequency content. Figure 10.2 shows how a coarse volume model can be combined with the volume perturbation technique to produce an extremely detailed interactively rendered cloud. The original 64^3 voxel data set is generated from a simple combination of volumetric blended implicit ellipses and defines the cloud macrostructure, as described in Chapter 9. The final rendered image in Figure 10.2(c), produced with the volume perturbation

Team LRN



(a)

(b)

(c)

FIGURE 10.2 Procedural clouds generated at interactive rates by Joe Kniss: (a) the underlying data (64^3); (b) the perturbed volume; (c) the perturbed volume lit from behind with low-frequency noise added to the indirect attenuation to achieve subtle iridescent effects.

Team LRN

FIGURE 10.3 More complex cloud scene rendered at interactive rates using a two-level approach. Image created by Joe Kniss.

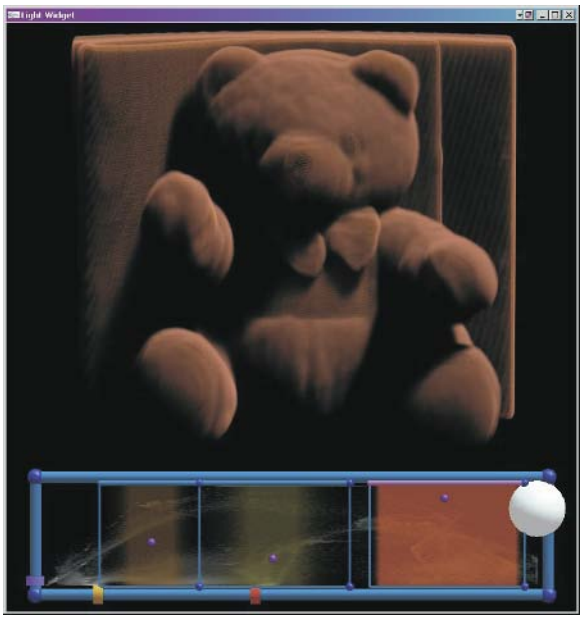
technique, shows detail that would be equivalent to an unperturbed voxel data set of at least one hundred times the resolution. Figure 10.3 contains another cloud image created with the same technique that shows more complex cloud structures with dramatic lighting. Figure 10.4 demonstrates this technique on another example. By perturbing the volume data set of a teddy bear with a high-frequency noise, a furlike surface on the teddy bear can be obtained.

These volumetric procedural models of clouds and fur can be animated by updating the texture coordinates of the noise texture octaves each frame. Dynamically changing three-dimensional textures is too time consuming; however, varying perturbation amounts and animation of texture coordinates offers a fast, effective way to animate procedural techniques in real time.

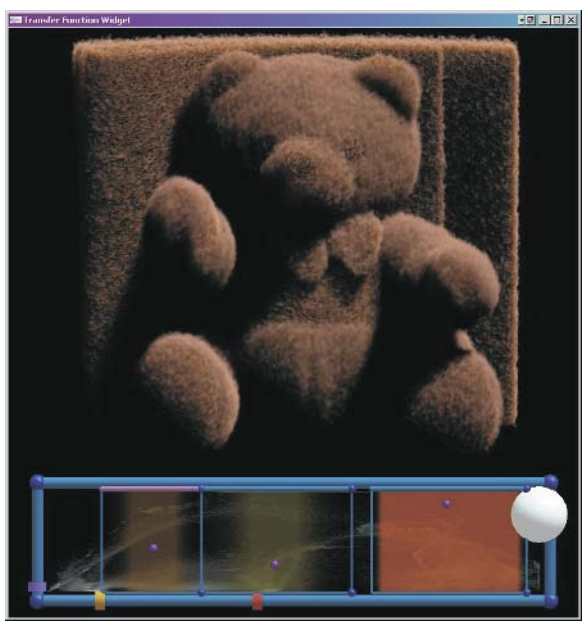
CONCLUSION

This chapter discussed a number of important issues related to adapting procedural techniques to real-time graphics hardware. As graphics hardware and CPU hardware change, the trade-offs between computation in the CPU and on the GPU will vary. However, issues such as flexibility, storage, levels of computation, and language interfaces will persist. Designing effective real-time procedural techniques is a challenge that has started to become solvable. The examples in this chapter should give some insight into approaches that can be used to adapt procedural techniques to real-time rendering.

Team LRN



(a)



(b)

FIGURE 10.4 Volume renderings of a teddy bear volume data set by Joe Kniss: (a) the original volume; (b) procedural high-frequency fur added in the rendering process.

11

PROCEDURAL SYNTHESIS OF GEOMETRY

JOHN C. HART

This chapter focuses on procedural methods that generate new graphical objects by synthesizing geometry. These methods have been designed to model the intricate detail found in natural shapes like plants. Unlike the shader, which is treated as an operator that can deform the geometry of some base reference object, procedural geometry synthesis creates an entirely new object by generating its geometry from nothing.

The ideas of procedural geometry synthesis extend naturally to procedural scene synthesis. Not only are these techniques capable of growing a tree from a sprout, but they can also populate an entire forest with unique instances of the tree (see Figure 11.1) (Deussen et al. 1998). They can also be applied to other, nonbiological applications, such as the automatic synthesis of entire cities of unique buildings (Parish and Müller 2001).

The most popular method for describing procedural models of plants and natural shapes has been the L-system (Prusinkiewicz and Lindenmayer 1990). The L-system is a grammar of replacement rules that capture various self-similar aspects of biological shape and development. The grammar operates on a set of turtle graphics symbols that translate the words generated by the grammar into graphical objects. The next section briefly reviews the L-system and turtle graphics and demonstrates them with a few samples.

Algorithms for the synthesis of procedural geometry fall into two classes. Data amplification algorithms evaluate procedures to synthesize all of their geometry; lazy evaluation algorithms evaluate the geometry synthesis procedures on demand. These paradigms each have benefits and drawbacks, which we will compare later in this chapter.

While the L-system, coupled with turtle graphics, has been an effective and productive language for articulating a large variety of natural and artificial shapes and scenes, it does suffer two shortcomings.

Team LRN

FIGURE 11.1 A procedurally synthesized environment where each plant species was ray-traced separately and composited into the final scene. The renderer executed a plant modeling program called `xfrog` to generate the plants on demand, using a lazy evaluation technique described later in the chapter. Image courtesy of Oliver Duessen. Copyright © 1998 Association for Computational Machinery, Inc.

Procedural techniques are used to model detail, but the efficient processing and rendering of scenes containing a large amount of geometric detail require it to be organized into a hierarchical, spatially coherent data structure. Turtle graphics objects are specified by a serial stream of instructions to the turtle, which, in many cases, is not the most effective representation for processing and rendering.

Furthermore, L-systems are difficult to process by humans. The L-system works with a large variety of symbols to specify operations within a full-featured 3D turtle-based scene description language. While these symbols are mnemonic, they can be likened to an assembly language. Their density makes them difficult to visually parse, and the large number of symbols used makes their articulation cumbersome.

Team LRN

The scene graph used by most commonly available modeling systems can be used to perform some limited procedural geometry synthesis. The replacement rules in an L-system that allow it to model self-similar detail can be represented in the scene graph using instancing. We will investigate the use of the scene graph for procedural geometry synthesis and demonstrate its abilities on a few simple procedural models.

The scene graph has some major benefits as a technique for procedural geometry synthesis. Its models can more easily be organized into hierarchical, spatially coherent data structures for efficient processing and rendering. Moreover, the articulation of scene graphs is more familiar to the computer graphics community, as demonstrated by the variety of scene description languages and programming libraries based on it.

But the standard form of the scene graph available in graphics systems is too limited to represent the wide variety of shapes and processes modeled by L-systems. We will show that an extension of the capabilities of the scene graph, called *procedural geometric instancing*, allows it to represent a larger subset of the shapes that can be modeled by an L-system. This subset includes a wide variety of the natural detailed shapes needed by typical production graphics applications.

THE L-SYSTEM

The L-system is a string rewriting system invented by Aristid Lindenmayer (1968). It was later shown to be a useful tool for graphics by Alvy Ray Smith (1984). Przemyslaw Prusinkiewicz then led an effort to develop it into a full-featured system for modeling the behavior of plant growth (Prusinkiewicz and Lindenmayer 1990).

An L-system is a grammar on an alphabet of symbols, such as "`F`", "`+`", and "`-`". The development of an L-system model is encapsulated in a set of *productions*—rules that describe the replacement of a nonterminal symbol with a string of zero or more symbols. For example, a production might be

$$\texttt{F} \rightarrow \texttt{F+F--F+F} \tag{11.1}$$

which says that every `F` in the input string should be replaced by `F+F--F+F`, which increases the number of nonterminal symbols in the string, causing it to grow with each production application.

Team LRN

This L-system growth is seeded with an *axiom*—an initial string such as the singleton `F`. Applying the production turns the initial string into `F+F--F+F`. Applying it again to this result yields

```
F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F
```

with spaces inserted to aid readability.

Unlike context-free grammars, L-system productions are applied in parallel, replacing as many symbols as possible in the current result. Hence the output of an L-system does not depend on the order the productions are applied.

This is a deterministic context-free L-system (a.k.a. a D0L-system), because each nonterminal symbol is the trigger of only one production, and this production does not care what symbols are to the left or right of the nonterminal symbol. An L-system can also be context-sensitive such that a nonterminal symbol would be replaced by something different depending on its surrounding symbols.

The symbols that an L-system operates on are assigned a geometric meaning. One such assignment is to interpret the strings as instructions to a turtle impaled with a pen (Abelson and diSessa 1982). The symbol "`F`" moves the turtle forward (leaving a trail) and the symbol "+" (respectively "`-`") rotates the turtle counterclockwise (clockwise) by a preset angle. Using the results of the example L-system and setting the angle of rotation to ± 60° yields the shapes shown in Figure 11.2.

In this manner, we can make a large variety of squiggly lines. We want to use the L-system, however, to model plant structures. Plant structures use branching as a

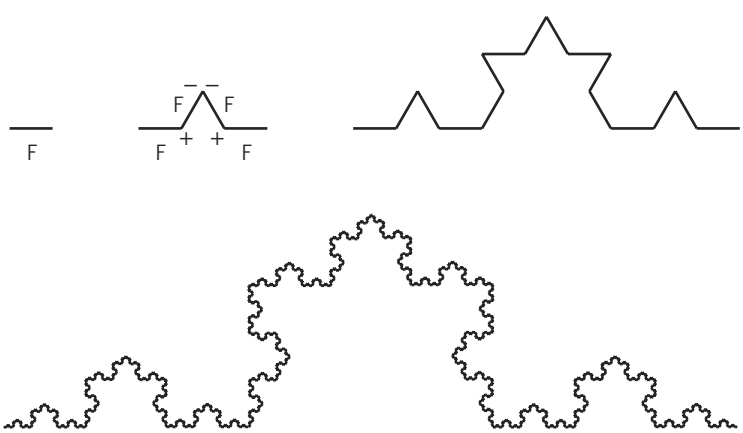


FIGURE 11.2 The von Koch snowflake curve development modeled by the L-system (11.1) and the shape to which it eventually converges.

means of distributing leaves that collect photosynthetic energy across a wide area, while concentrating the flow of nutrients to and from a similarly branching root system through a single trunk.

When a sequence of symbols generated by the L-system completes the description of a branch, the turtle must return to the base of the branch to continue describing the remainder of the plant structure. The L-system could be cleverly designed to allow the turtle to retrace its steps back to its base, but it would be cumbersome, inefficient, and incomprehensible. A better solution is offered through the use of brackets to save and restore the state of the turtle. The left bracket symbol "`[`" instructs the turtle to record its state (position and orientation); the right bracket "`]`" instantly teleports the turtle to its most recently stored state. This state information can be stored on a stack, which allows bracket symbols to be nested.

These state store/restore commands allow L-systems to model branching structures. For example, the L-system

$$\texttt{F} \rightarrow \texttt{F[+F]F[-F]F} \tag{11.2}$$

grows the sprout `F` into the trees shown in Figure 11.3.

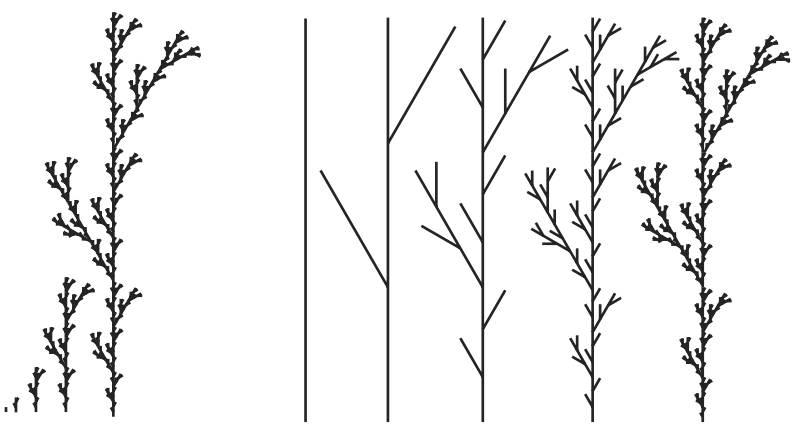


FIGURE 11.3 Development of the simple tree modeled by the L-system (11.2). (a) In this progression, the segment length remains constant, which simulates the growth of the bush from a sapling. (b) In this progression, the length of the segments decreases by 1/3 in each iteration and the tree increases its detail in place. While the progression in (a) is a developmental model useful for simulating the biological and visual properties of the growth of the tree, the progression in (b) represents a hierarchical model useful in efficiently processing and rendering the tree.

Team LRN

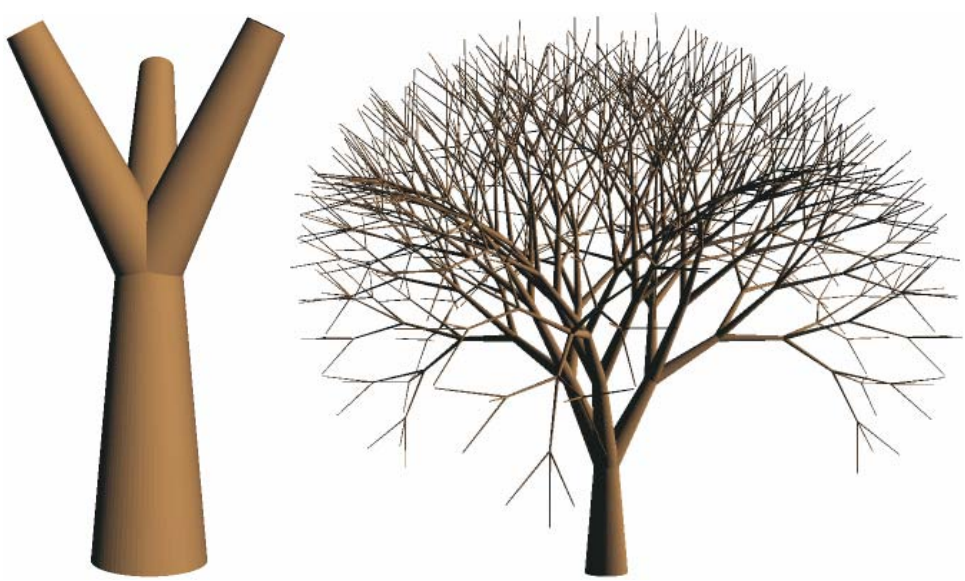
The turtle graphics shape description has been extended into a full-featured three-dimensional graphics system. In three dimensions the symbols "+" and "-" change the turtle's *yaw,* and the new symbols "^" and "&" change the turtle's pitch (up and down), whereas "\" and "/" change its roll (left and right). We can use these 3D turtle motions to grow a ternary tree with the L-system production

```
F → F[&F][&/F][&\F]
```

which attaches three branch segments to the end of each segment, shown in Figure 11.4 along with the resulting tree after several iterations of the production.

In these examples, the segment lengths and rotation angles have all been identical and fixed to some global constant value. When modeling a specific growth structure, we may want to specify different segment lengths and rotation angles. The elements of the turtle geometry can be parameterized to provide this control. For example, `F(50)` draws a segment of length 50 units, and `+(30)` rotates the turtle 30° about its yaw axis. Note that using a 3D turtle, Figure 11.2 can be constructed by the production

```
F → /(180)-(30)F+(60)F-(30)\(180)
```



(a) (b)

FIGURE 11.4 (a) The trunk and (b) the resulting ternary tree modeled using 3D turtle commands.

Team LRN

which constructs the snowflake curve out of two smaller inverted snowflake curves. The curves are inverted by flipping the turtle over, which exchanges counterclockwise with clockwise.

In three dimensions, the line segments are replaced with cylinders. These cylinders represent the branches, twigs, and stems of the plant structure. Newer branches should be thinner than older branches near the base. The symbol "`!`" decreases the width of all segments specified after it in a string. Note that the "`]`" bracket resets the branch diameter to the state corresponding to the location of the "`[`" symbol in the string. Hence "`F[!+F]F`" draws a trunk of two equal-diameter segments with a thinner segment branching off its side. The segment length can also be specified directly. The "`!(`*d*`)`" substring sets the current segment diameter to *d*.

The L-system approximates the shape of the plant support structure as a sequence of straight cylinder segments. These cylinder segments can be joined by placing a sphere at the joints. The turtle command "`@O`" draws a sphere around the current turtle position whose diameter is equal to the current segment diameter. Hence "`F@O+F`" uses a sphere as an elbow between the two cylinders. Alternatively, a generalized cylinder can be constructed as a tube around a space curve by using the symbol "`@Gc`" to specify the current turtle position as a control point, and using the symbols "`@Gs`" and "`@Ge`" to start and end the curve segment. The diameter of the generalized cylinder can also be set differently at each control point using the "`!`" symbol before each "`@Gc`" substring.

The turtle geometry system quickly ran out of symbols from the ASCII character set and so needs multicharacter symbols. These multicharacter symbols are usually prefixed by the "`@`" character.

Polygons can also be defined using the turtle geometry. The turtle motion inside the delimiters "`{`" and "`}`" is assumed to describe the edges of a filled polygon. Within these braces, the symbol "`F`" describes an edge between two vertices. That operation can be decomposed into the "`.`" symbol, which defined a vertex at the current turtle position, and the "`G`" symbol, which moves the turtle without defining a vertex.

Bicubic patches can also be modeled. The substring "`@PS(`*n*`)`" defines a new bicubic patch labeled by the integer *n*. The substring "`@PC(`*n*`,`*i*`,`*j*`)`" defines the position of the (*i*,*j*) control point to the current turtle position. The substring "`@PD(`*n*`)`" indicates that patch *n* should be drawn.

Color and texture can also be specified. Colors are predefined in a color map, and the symbol "`;(`*f*`,`*b*`)`" defines the frontside color to index *f* and the backside color to index *b*. The symbol "`;`" alone increments the current color index. For example, a three-element color table containing brown, yellow, and green can be used for the colors of branches, twigs, and stems in the string "`F[;+F[;+F]`

Team LRN

`[;-F]][;-F[+F][-F]]`". Note that the color index is reset by the "`]`" closing bracket. The symbol "`@T(`*n*`)`" can be used to indicate that the output of subsequent turtle commands should be texture mapped. The parameter *n* indicates which texture map should be used. Texture mapping is turned off by the "`@T(0)`" substring.

The command "`@L(`*label*`)`" prints the label at the current turtle location, which is useful for debugging or annotation. The command "`@S(`*command*`)`" executes an operating system command, which could be used, for example, to provide audio feedback during the L-system processing.

PARADIGMS GOVERNING THE SYNTHESIS OF PROCEDURAL GEOMETRY

Many different interactive modeling systems tend to follow the same standard flow of data (e.g., Figures 1.5, 3.2, and 7.3 of Foley et al. 1990). The *user* articulates a conceptual model to the modeler. The *modeler* interprets the articulation and converts it into an intermediate representation suitable for manipulation, processing, and rendering. The *renderer* accepts the intermediate representation and synthesizes an image of the object. The user then observes the rendered model, compares it to the conceptual model, and makes changes according to their difference.

The procedural synthesis of geometry follows this same data flow model, although the implementation of the specifics of the intermediate representation can follow two different paradigms: data amplification or lazy evaluation.

Data Amplification

The L-system describes procedural geometry following the paradigm of a *data amplifier,* as shown in Figure 11.5. Smith (1984) coined this term to explain how procedural methods transform the relatively small amounts of information in a procedural model articulation into highly detailed objects described by massive amounts of geometry. (The figure's representation of the data amplifier with an op-amp circuit is not far fetched. The electronic amplifier scales its output voltage and

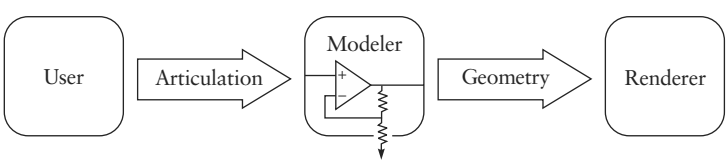


FIGURE 11.5 The "data amplifier" procedural modeling system.

then combines it with the input voltage. Such rescale-and-add processes form the basis of a variety of procedural models.)

Procedural modelers that follow the data amplification paradigm synthesize some kind of intermediate geometric representation. This intermediate representation is a scene description consisting of polygons or other primitives. For an L-system model of a tree, the intermediate representation would consist of cones representing the branches and polygonal meshes representing the leaves. The intermediate representation is then passed to a renderer to convert it into an image.

Procedural methods are used to synthesize detailed scenes. The intermediate representation for such scenes can become extremely large, growing exponentially in the number of L-system production applications. For example, the L-system description for a single poplar tree was only 16 KB, but when evaluated yielded a 6.7 MB intermediate representation (Deussen et al. 1998). Data amplification causes an intermediate data explosion as a compact procedural model is converted into a huge geometric representation, which is then rendered into an image.

The intermediate representation generated by an L-system is also largely unorganized. The geometry output by evaluating an L-system is in a stream order that follows the path of the turtle. The bracketing used to indicate branching structures does organize this stream into a postorder traversal, but this organization has not been utilized. The lack of hierarchical organization prevents renderers from capitalizing on spatial coherence when determining visibility. Standard methods can process the unorganized geometry into more efficient data structures such as a grid or octree. However, these data structures further increase the storage requirement.

We can avoid storage of the intermediate geometry representation by rendering the primitives generated by the turtle as soon as they are generated. This avoids the intermediate data explosion, but at the expense of the extra time required to manage the turtle during the rendering process. This also limits the rendering to per-polygon methods such as Z-buffered rasterization and would not work for per-pixel methods such as ray tracing.

Reeves and Blau (1985) rendered the turtle graphics generated by an L-system using particles instead of geometry. The particles were rendered directly to the screen as filtered streaks of shading. The drawback of rendering particle trails directly to the frame buffer is indirectly due to aliasing. The streaks of shading resulting from a rendered particle trail represent fine details, such as pine needles or blades of grass. These details often project to an area smaller than the size of a pixel and use the alpha channel to contribute only a portion of the pixel's color. For this technique to work correctly in a Z-buffered rendering environment, the particle traces must be rendered in depth order, and the particle traces must not intertwine. Fortunately, a

Team LRN

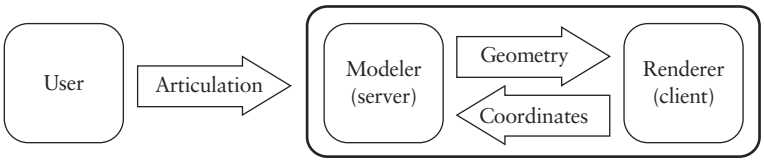


FIGURE 11.6 The "lazy evaluation" procedural modeling system.

precisely correct rendering is often not necessary. Minor occlusion errors are easily obfuscated by the overwhelming amount of visual detail found in procedurally synthesized scenes. "The rich detail in the images tends to mask deviations from an exact rendering" (Reeves and Blau 1985, p. 313).

Another option is to rasterize the streaks of shading generated by the particle traces into an intermediate volume (Kajiya and Kay 1989). Direct volume-rendering techniques can be used to render the intermediate representation to more accurately determine visibility. This technique was used to simulate the shading of fur and carpeting, and modeled a convincing teddy bear.

Lazy Evaluation

Lazy evaluation avoids intermediate geometry representation problems of the data amplifier by executing the geometry synthesis procedure only when it is needed. Lazy evaluation is illustrated diagrammatically in Figure 11.6.

Lazy evaluation facilitates a client-server relationship between the modeler and the renderer, allowing the modeler to generate only the geometry needed for the renderer to draw an accurate picture. This process saves time and space by avoiding the need to store and process a massive intermediate geometric representation.

In order for lazy evaluation to work in a procedural geometry system, the renderer needs to know how to request the geometry it needs, and the procedural model needs to know what geometry to generate. This process creates a dialog between the modeler and the renderer.

Procedural geometries can sometimes be organized in spatially coherent data structures. Examples of these data structures include bounding volume hierarchies, octrees, and grids. Such data structures make rendering much more efficient. The ability to cull significant portions of a large geometric database allows the renderer to focus its attention only on the visible components of the scene. Techniques for capitalizing on spatial coherence data structures exist for both ray tracing (Rubin and Whitted 1980; Kay and Kajiya 1986; Snyder and Barr 1987) and Z-buffered rasterization (Greene and Kass 1993).

Team LRN

These spatial coherence data structures also support lazy evaluation of the procedural model. For example, the modeler can generate a bounding volume and ask the renderer if it needs any of the geometry it might contain. If the renderer declines, then the procedural model does not synthesize the geometry inside it. The tricky part of implementing the lazy evaluation of a procedure is determining the bounding volume of its geometry without actually executing the procedure.

L-systems alone do not contain the necessary data structures to support lazy evaluation. The next section describes how L-systems can be simulated with scene graphs, which include the data structures and algorithms that support lazy evaluation.

THE SCENE GRAPH

Geometric objects are often constructed and stored in a local "model" coordinate system specific to the object. These objects may be placed into a scene by transforming them into a shared "world" coordinate system. The placement of a model into a scene is called *instancing*. The model is called the *master,* and the transformed copy placed in a scene is called an *instance* (Sutherland 1963).

In order to manage complex scenes containing many objects, objects can be organized into hierarchies. Instances of individual objects can be collected together into a composite object, and the composite object can be instanced into the scene. The structure of this hierarchy is called a *scene graph*.

There are a variety of libraries and languages available to describe scene graphs. For example, the Java3D and Direct3D graphics libraries contained a "retained mode" scene graph that allowed entire scenes to be stored in memory and rendered automatically. The VRML (Virtual Reality Modeling Language) is a file representation of a scene graph that can be used to store a scene graph as a text file to be used for storage and communication of scenes.

An instance is a node in a scene graph consisting of a transformation, a shader, and a pointer to either another scene graph node or a geometric primitive, as shown in Figure 11.7(a). The transformation allows the instance to be transformed by any affine transformation, and the shader allows the instances of an object to have different appearance properties. Instancing can turn a tree-structured scene graph into a directed acyclic graph, such that a node may have more than one parent in the hierarchy, as shown in Figure 11.7(b).

In the following examples, we use a specialized scene description language that encourages instancing. Scene graph nodes are named and can be accessed by their given name. An object corresponding to a scene graph node is described using the syntax

Team LRN

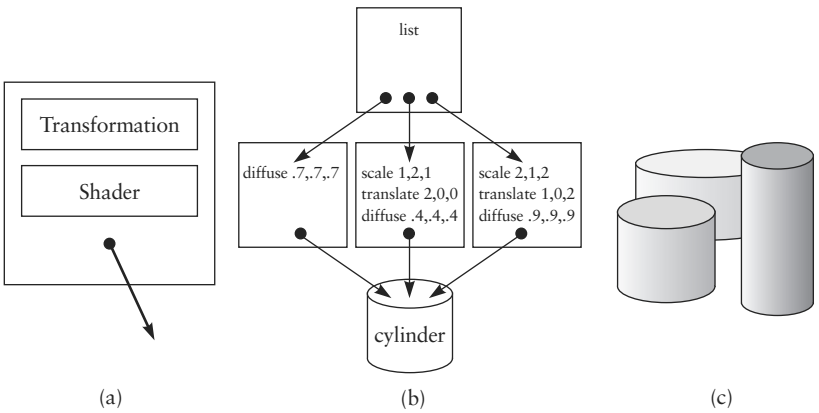


FIGURE 11.7 Anatomy of an instance: (a) The instance is a pointer to another object, with additional transformation and shading parameters. (b) The scene graph is a collection of instances, in this case a list of three instances of a cylinder primitive. (c) The configuration of the cylinders.

```
define <name> <description> end
```

where `<description>` is a list of instances and/or primitives. Instances and primitives not contained as named nodes are assumed to be elements of an implicit list node at the top of the scene graph.

Each instance or primitive precedes a list of zero or more transformation and shading commands. The transformation and shading commands are accumulated in order and act only on the most recently specified instance or primitive. When a new instance or primitive is declared, the transformation is reset to the identity, and the shading is restored to the default shading of the current parent object. Hence the specification

```
cylinder(10,1) scale 2
cylinder(10,1) rotate 30,(0,0,1)
```

creates the union of a cylinder along the y-axis of length 10 and radius 1 uniformly scaled by two, and a second cylinder also of length 10 and radius 1 rotated by 30° about the z-axis. The second cylinder is not scaled by two.

The transformation and shading commands of an instance are applied relative to the current transformation and shading of the context into which they are instanced. For example, the description

Team LRN

```
define A
    sphere translate (2,0,0) diffuse 1,(1,0,0)
end

define B
    A rotate 90,(0,0,1) diffuse 1,(0,0,1)
end
```

defines object A as a red unit sphere centered at the point (2,0,0), and object B as a blue sphere centered at the point (0,2,0).

The scene graph can use instancing to implement turtle graphics. For example, the turtle graphics stream "F[+F]F[-F]F" can be implemented in our scene description language as

```
define F
    cylinder(1.0,0.1)
end

define +F
    F rotate 30,(0,0,1)
end

define -F
    F rotate -30,(0,0,1)
end

F scale 1/3
+F scale 1/3 translate (0,1/3,0)
F scale 1/3 translate (0,1/3,0)
-F scale 1/3 translate (0,2/3,0)
F scale 1/3 translate (0,2/3,0)
```

One problem worth noting is that the state after the turtle has moved and must be maintained. This is why the translation commands must be inserted after some of the instances.

We can also use instancing to implement the productions of an L-system (Hart 1992). Consider the single-production L-system

$$A \rightarrow F[+F]F[-F]F \tag{11.3}$$

with the axiom "A". We can represent this structure as a scene graph as

```
define A
    F scale 1/3
    +F scale 1/3 translate (0,1/3,0)
    F scale 1/3 translate (0,1/3,0)
```

Team LRN

```
    -F scale 1/3 translate (0,2/3,0)
    F scale 1/3 translate (0,2/3,0)

end

A
```

where objects "F","+F", and "-F" are defined as before. An equivalent scene graph is shown in Figure 11.8(a), which yields the arrangement of cylinders shown in Figure 11.9(a).

This is even more powerful when we represent an additional iteration of development as the L-system

$$A \rightarrow F[+F]F[-F]F \tag{11.4}$$

$$B \rightarrow A[+A]A[-A]A \tag{11.5}$$

with the axiom "B". The corresponding scene graph is now

```
define +A A rotate 30,(0,0,1) end
define -A A rotate -30,(0,0,1) end

define B
    A scale 1/3
    +A scale 1/3 translate (0,1/3,0)
    A scale 1/3 translate (0,1/3,0)
```

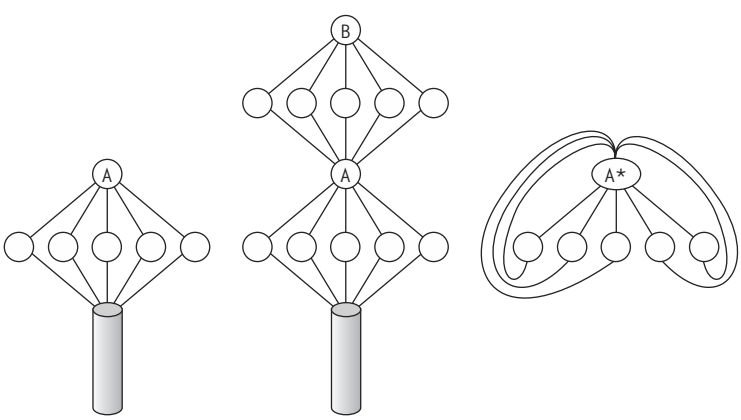


FIGURE 11.8 Scene graphs for the bush L-system: (a) one iteration, (b) two iterations, and (c) infinitely many iterations.

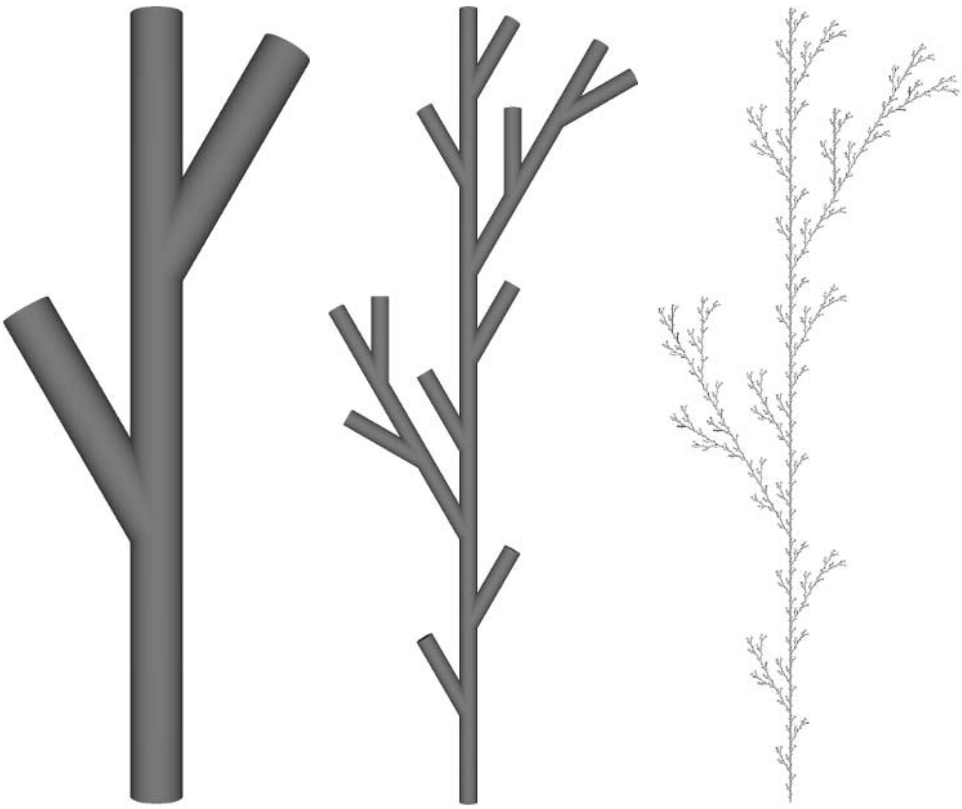


FIGURE 11.9 Bushes modeled by (a) "A", (b) "B", and (c) "A*".

```
    -A scale 1/3 translate (0,2/3,0)
    A scale 1/3 translate (0,2/3,0)
end

B
```

where "A" is defined as before. This scene graph is equivalent to the one shown in Figure 11.8(b) and yields the arrangement of cylinders shown in Figure 11.9(b).

Noticing the similarities between "A" and "B", you might be tempted to develop the instance

```
define +A* A* rotate 30,(0,0,1) end
define -A* A* rotate -30,(0,0,1) end
```

Team LRN

```
define A*
    A* scale 1/3
    +A* scale 1/3 translate (0,1/3,0)
    A* scale 1/3 translate (0,1/3,0)
    -A* scale 1/3 translate (0,2/3,0)
    A* scale 1/3 translate (0,2/3,0)
end
```

where the scale is introduced to keep the results the same size. This scene graph is equivalent to the one shown in Figure 11.8(c) and yields the limit set shown in Figure 11.9(c).

A cyclic scene graph describes the often (but not always) fractal shape that is the limit case of the L-system. The scene graph now has cycles and consists entirely of instances (no primitives), which means the resulting shape is described entirely by transformations, without any base geometry. Such structures are called (*recurrent*) *iterated function systems*, and special rendering techniques need to be used to display them (Hart and DeFanti 1991).

Figure 11.10 shows another example of the structure of scene graphs. The tops of each of the scene graphs all consist of the same union (list) node of three elements. These elements are instance nodes whose transformations are indicated by

- `A`—scale by 1/2 and translate (−0.433,−0.25,0)
- `B`—scale by 1/2 and translate (0,0.5,0)
- `C`—scale by 1/2 and translate (−0.433,−0.25,0)

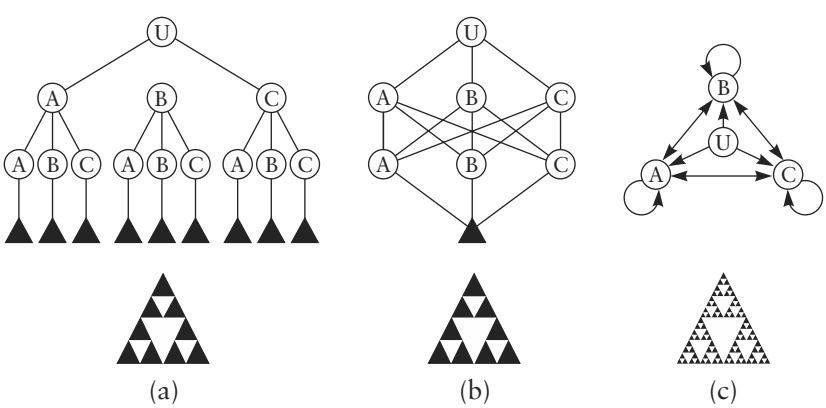


FIGURE 11.10 (a) Tree-structured scene graph, (b) directed acyclic scene graph, and (c) cyclic scene graph. The resulting scene is shown below each scene graph.

The scene graph in Figure 11.10(b) compactly combines the common subtrees of the scene graph in Figure 11.10(a). The scene graph in Figure 11.10(b) contains two copies of each node, which could be further compacted into the scene graph in Figure 11.10(c) if cycles are allowed. However, adding cycles results in a (recurrent) iterated function system, which describes objects entirely by transformations with no explicit geometric primitives (e.g., the triangle primitives are missing from the scene graph in Figure 11.10(c)).

We would like to get the power of cycles shown in Figure 11.10(c), but without the loss of geometric primitives or the difficulty in detecting cycles. We would instead like to permit conditional cycles that could terminate under some resolution-specific criterion.

The scene graph representation also provides a clear hierarchical organization of the procedural geometry for more efficient processing and rendering using lazy evaluation. A bounding volume can be stored at each of the instance nodes, and this bounding volume can be used to determine if a node's underlying geometry need be generated. For example, the earlier iterates of Figure 11.9 can serve as bounding volumes for the later iterates.

The scene graph is only capable of representing a small subfamily of L-systems, specifically deterministic nonparametric context-free L-systems without global effects such as tropism. In order to convert the more powerful extensions of L-systems into more efficient scene graphs, the scene graph will need to be augmented with the procedural extensions described in the next section.

PROCEDURAL GEOMETRIC INSTANCING

Procedural geometric instancing (PGI) augments the instance in a scene graph with a procedure, as shown in Figure 11.11. This procedure is executed at the time of instantiation (i.e., every time the object appears in a scene). The procedure has access to the instance node that called it and can change the node's transformation and shading parameters. The procedure can also change the instance to refer to a different object node in the scene graph. In addition, the procedure also has access to external global scene graph variables, such as the current object-to-world coordinate transformation matrix and passing parameters between nodes.

Parameter Passing

One enhancement to the L-system model allows it to describe shapes using real values (Prusinkiewicz and Lindenmayer 1990). Among other abilities, parametric L-systems can create more complex relationships between parent and child geometries.

Team LRN

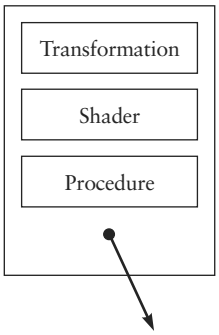


FIGURE 11.11 A procedural instance. The procedure is executed at the time of instantiation.

Parameters are by no means new to the concept of instancing. For example, they are present in SPHIGS (Foley et al. 1990), but their use here to control hierarchical subdivision differentiates them from previous implementations. Parameters are bound at instantiation in the standard fashion using a stack to allow recursion. Parameters may alter transformations or may be passed along to further instances. Both uses are demonstrated in the following example.

Example: *Inductive Instancing*

Iterative instancing builds up vast databases hierarchically, in $O(\log n)$ steps, by instancing lists of lists [Snyder and Barr 1987; Hart 1992]. For example, a list of four instances of a blade of grass makes a `clump`, four `clumps` make a `patch`, and four `patches` make a `plot`—of 64 blades of grass. Though each list consists of four instances of the same element, each instance's transformation component differs to describe four distinct copies of the geometry.

Even the step-saving process of iterative instancing becomes pedantic when dealing with billions of similar objects. Inductive instancing uses instancing parameters to reduce the order of explicit instancing steps in the definition from $O(\log n)$ to $O(1)$. Using the field-of-grass example, we define an object `grass(n)` as a list of four instances of `grass(n-1)`, and end the induction with the definition of `grass(0)`, which instances a single blade of grass.

```
define grass(0) blade end

define grass(n)
    grass(n-1)
    grass(n-1) translate 2^n*(0.1,0.0,0.0)
```

Team LRN

```
    grass(n-1) translate 2^n*(0.0,0.0,0.1)
    grass(n-1) translate 2^n*(0.1,0.0,0.1)
end

grass(15)
```

Hence, a single instance of `grass(i)` inductively produces the basis of the scene in Figure 11.12 containing 4^i blades of grass.

Inductive instancing is similar in appearance to the predicate logic of Prolog. Organized properly, the defined names may be compared against the calling instance name until a match is found. For example, `grass(15)` would not match `grass(0)` but would match `grass(`*n*`)`.

Accessing World Coordinates

Objects are defined in a local coordinate frame, but are instanced into a world coordinate system. In some situations, an instance may need to change its geometry based on its global location and orientation. Given an object definition and a specific instantiation, let W denote the 4×4 homogeneous transformation matrix that maps the object to its instantiation. The transformation W maps local coordinates to world coordinates.

Procedural geometric instancing adopts the convention that within the scope of an instance, the object-to-world transformation that is available to the procedure is the one from the beginning of the instantiation and is unaffected by the instance's transformations. This solves an ordering problem where a scale followed by a

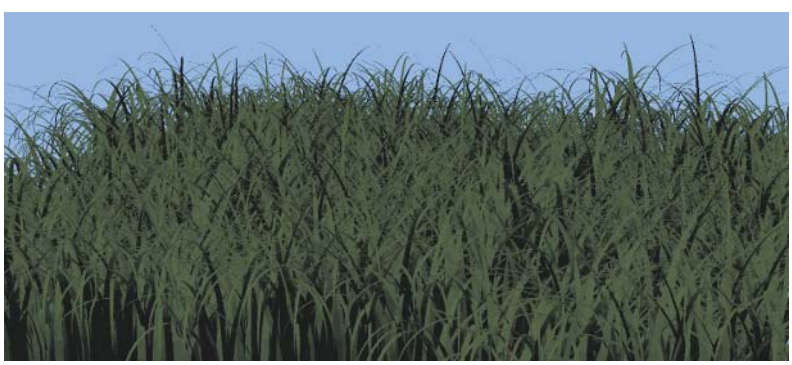


FIGURE 11.12 Grass modeled through iterative instancing, with randomness added using techniques described later in this section.

Team LRN

FIGURE 11.13 These trees have equivalent instancing structures except that the one on the right is influenced by downward tropism, simulating the effect of gravity on its growth.

rotation (which ordinarily are mutually commutative) might not be equivalent to a rotation followed by a scale if either depends on global position or orientation.

The following three examples demonstrate procedural models requiring access to world coordinates.

Example: *Tropism*

L-systems simulate biological systems more realistically through the use of global effects. One such effect is *tropism*—an external directional influence on the branching patterns of trees (Prusinkiewicz and Lindenmayer 1990). Downward tropism simulates gravity, resulting in sagging branches; sideways tropism results in a wind-blown tree; and upward tropism simulates branches growing toward sunlight. In each case, the tropism direction is uniform, regardless of the local coordinate system of the branch it affects.

Given its global orientation, an instance affected by tropism can react by rotating itself into the direction of the tropism. For example, the following PGI specification models Figure 11.13, illustrating two perfectly ternary trees, although one is made more realistic through the use of tropism.

```
define limb(r, l) cone(l,r,0.577*r) end
define tree(0, r, l, t, alpha)
    limb(r,l)
    leaf translate (0,l,0)
end
```

Team LRN

```
define branch(n, r, l, t, alpha)
    tree(n,r,l,t,alpha) rotate 30,(0,0,1)
end

define tree(n, r, l, t, alpha)
    limb(r,l)
    branch(n-1,0.577*r,0.9*l,t,alpha)
        tropism((0,1,0,0),-alpha,t)
        translate (0,l,0)
    branch(n-1,0.577*r,0.9*l,t,alpha)
        tropism((0,1,0,0),-alpha,t)
        rotate 120,(0,1,0) translate (0,l,0)
    branch(n-1,0.577*r,0.9*l,t,alpha)
        tropism((0,1,0,0),-alpha,t)
        rotate 240,(0,1,0) translate (0,l,0)
end

tree(8,.2,1,(0,-1,0),0) translate(-4,0,0)
tree(8,.2,1,(0,-1,0),20) translate(4,0,0)
```

The procedure `tropism` is defined as

$$\texttt{tropism}(\mathbf{v}, \alpha, \mathbf{t}) \equiv \texttt{rotate}\ \alpha\ ||W\mathbf{v} \times \mathbf{t}||, W\mathbf{v} \times \mathbf{t} \tag{11.6}$$

This `tropism` definition may be substituted by hand, translated via a macro, or hard-coded as a new transformation.

The object `limb` consists of an instance of `tree` rotated 30°. Under standard instancing, this object could be eliminated, and the 30° could be inserted in the definition of `tree`. However, this operation is pulled out of the definition of `tree` because the world transformation matrix W used in the definition of `tropism` is only updated at the time of instantiation. The separate definition causes the tropism effect to operate on a branch after it has rotated 30° away from its parent's major axis.

Tropism is typically constant, although a more accurate model would increase its severity as branches become slimmer. Thompson (1942) demonstrates that surface tension dictates many of the forms found in nature. In the case of trees, the strength of a limb is proportionate to its surface area, $l \times r$, whereas its mass (disregarding its child limbs) is proportionate to its volume, $l \times r^2$. We can simulate this by simply increasing the degree of tropism α inversely with respect to the branch radius r. Hence, `branch` would be instantiated with the parameters

```
branch(n-1,0.577*r,0.9*l,t,(1-r)*alpha0)
```

where `alpha0` is a constant, maximum angle of tropism.

Team LRN

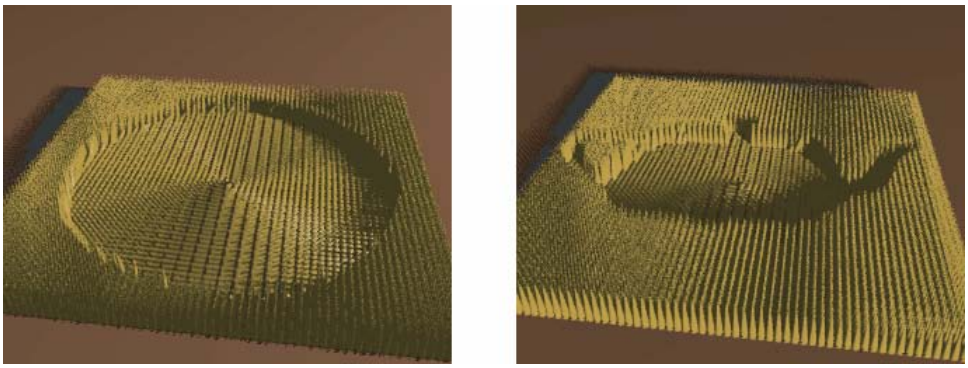


FIGURE 11.14 Crop circles trampled in a field of cones.

While variable tropism could be incorporated, via source code modification, as a new parameterized symbol in the turtle-based L-system paradigm, procedural geometric instancing provides the tools to articulate it in the scene description.

Example: *Crop Circles*

Prusinkiewicz, James, and Mech (1994) added a query command "?" to the turtle's vocabulary that returned the turtle's world position. This information was used to prune L-system development based on external influences. The world coordinate transformation can similarly detect the presence of an external influence, and the geometry can appropriately respond. A crop circle such as those allegedly left by UFO encounters is similar to the simulation of topiary, except that the response of pruning is replaced with bending as demonstrated in Figure 11.14.

Such designs can be described implicitly (Prusinkiewicz, James, and Mech 1994) in the case of a circle, or through a texture map (*crop* map) (Reeves and Blau 1985) in the case of the teapot.

Example: *Geometry Mapping*

The grass example from the previous section shows how geometry can be instanced an arbitrary number of times onto a given section of a plane. Consider the definition of a Bezier patch as a mapping from the parameter plane into space. This mapping

Team LRN

takes blades of grass from the plane onto the Bezier patch. Replacing the blades of grass with fine filaments of hair yields a fully geometric specification of fur or hair.

Other Functions

In addition to passing parameters and accessing world coordinates, several other features based on these abilities make specification of procedural models easier.

Random Numbers

Randomness can simulate the chaos found in nature and is found in almost all procedural natural modeling systems. Moreover, various kinds of random numbers are useful for natural modeling.

The notation `[a,b]` returns a random number uniformly distributed between `a` and `b`. The notation `{a,b}` likewise returns a Gaussian-distributed random number.

The Perlin *noise* function provides a band-limited random variable (Perlin 1985) and is implemented as the scalar-valued function `noise`. A typical invocation of the noise function using the world coordinate position is specified: `noise(W(0,0,0,1))`.

Example: *Meadows*

Fractional Brownian motion models a variety of terrain. This example uses three octaves of a $1/f^2$ power distribution to model the terrain of a hilly meadow. Grass is instanced on the meadow through a translation procedurally modified by the noise function. The placement of the grass is further perturbed by a uniformly random lateral translation, and its orientation is perturbed by the noise function.

The following PGI scene specification describes the meadow displayed in Figure 11.15. The vector-valued function `rotate(x,theta,axis)` returns the vector `x` rotated by `theta` about the axis `axis`. Its use in the definition of `fnoise` disguises the creases and nonisotropic artifacts of the noise function due to a simplified interpolation function.

```
#define NS 16 /* noise scale */
#define fnoise(x) (NS*(noise((1/NS)*(x)) +
                  0.25 noise((2/NS) rot((x),30,(0,1,0))) +
                  0.0625 noise((4/NS) rot((x),60,(0,1,0)))))
#define RES 0.1 /* polygonization resolution */
```

Team LRN

```
define plate(-1)
    polygon (-RES,fnoise(W(-RES,0,-RES,1)),-RES),
            (-RES,fnoise(W(-RES,0, RES,1)), RES),
            ( RES,fnoise(W( RES,0, RES,1)), RES)
    polygon ( RES,fnoise(W( RES,0, RES,1)), RES),
            ( RES,fnoise(W( RES,0,-RES,1)),-RES),
            (-RES,fnoise(W(-RES,0,-RES,1)),-RES)
end

define plate(n)
    plate(n-1) translate 2^n*( RES,0, RES)
    plate(n-1) translate 2^n*(-RES,0, RES)
    plate(n-1) translate 2^n*( RES,0,-RES)
    plate(n-1) translate 2^n*(-RES,0,-RES)
end

define blade(0) polygon (-.05,0,0),(.05,0,0),(0,.3,0) end

define blade(n)
    blade(n-1)
      scale (.9,.9,.9) rotate 10,(1,0,0) translate (0,.2,0)
    polygon (-.05,0,0),(.05,0,0),(.045,.2,0),(-.045,.2,0)
end

define grass(-2)
    blade(10)
      rotate 360*noise((1/16)*(W(0,0,0,1))),(0,1,0)
      translate [-.05,.05],fnoise(W(0,0,0,1)),[-.05,.05])
end
```

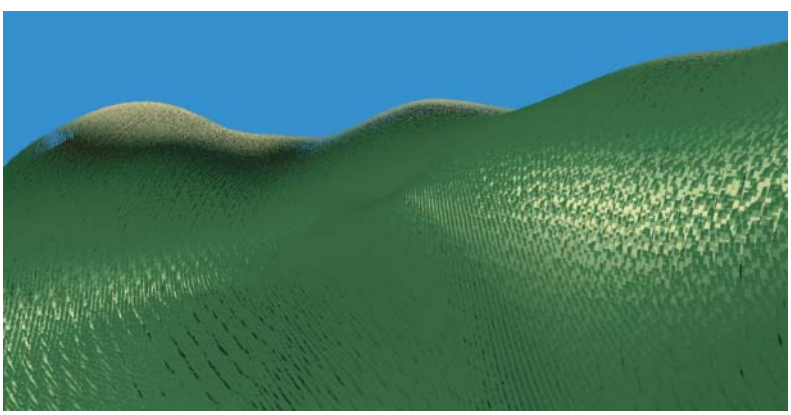


FIGURE 11.15 A grassy meadow with an exaggerated specular component to demonstrate reflection "hot spots." No texture maps were used in this image.

```
define grass(n)
    grass(n-1) translate 2^n*(.1,0,.1)
    grass(n-1) translate 2^n*(-.1,0,.1)
    grass(n-1) translate 2^n*(.1,0,-.1)
    grass(n-1) translate 2^n*(-.1,0,-.1)
end

plate(6)
grass(6)
```

Levels of Detail

The scale at which geometry projects to the area of a pixel on the screen under the rules of perspective is bound from above by the function

$$\texttt{lod}(\mathbf{x}) = ||\mathbf{x} - \mathbf{x}_0|| \, 2\tan(\theta/2)/n \tag{11.7}$$

where $\mathbf{x}_0$ is the eyepoint, θ is the field of view, and n is the linear resolution. The condition $\texttt{lod}(W(0,0,0,1)) > 1$ was used to halt recursive subdivision of fractal shapes constructed by scaled instances of the unit sphere (Hart and DeFanti 1991). In typical use, `lod` replaces complex geometries with simpler ones to optimize display of detailed scenes.

Comparison with L-Systems

L-systems are organized into families based on their representational power. The simplest family is the deterministic context-free L-system. Parameters were added to handle geometric situations requiring nonintegral lengths (Prusinkiewicz and Lindenmayer 1990). Stochasticism was added to simulate the chaotic influences of nature. Various degrees of context sensitivity can be used to simulate the transmission of messages from one section of the L-system model to another during development. Global influences affect only the resulting geometry, such as tropism, which can simulate the effect of gravitational pull when determining the branching direction of a tree limb.

Figure 11.16 depicts the representational power of procedural geometric instancing with respect to the family of L-system representations. Standard geometric instancing can efficiently represent only the simplest L-system subfamily, whereas there is currently no geometrically efficient representation for any form of context-sensitive L-system. Procedural geometric instancing is a compromise, efficiently representing the output of a stochastic context-free parametric L-system with global effects.

Team LRN

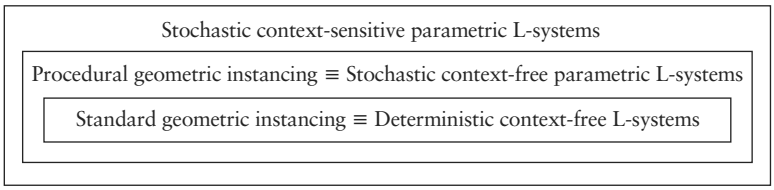

Stochastic context-sensitive parametric L-systems
Procedural geometric instancing ≡ Stochastic context-free parametric L-systems
Standard geometric instancing ≡ Deterministic context-free L-systems

FIGURE 11.16 Hierarchy of representations.

Ordering

Several rendering methods benefit from a front-to-back ordering of the geometry sent to it. We order our bounding volume hierarchy through the use of axially sorted lists. The contents of each list are sorted six times, in nonincreasing order of their most extreme points in the positive and negative x, y, and z directions. Each list is then threaded six times according to these orderings. At instantiation, we determine which of the six vectors, W^{-1T} $(\pm 1,0,0,0)$, W^{-1T} $(0,\pm 1,0,0)$, or W^{-1T} $(0,0,\pm 1,0)$, has the maximum dot product with a unit vector from the position W $(0,0,0,1)$ pointing toward the viewer. The elements of the list are then instantiated according to this order. While not perfect, this method does an effective job of instancing geometry globally in a front-to-back order.

BOUNDING VOLUMES

The bounding volume of procedural geometry must account for the different possible shapes the procedural geometry may take. A bounding volume is associated with every node in a PGI scene graph, and the procedure associated with the node can alter the bounding volume of the instantiation.

Early terrain models simulated fractional Brownian motion by midpoint displacement, which was an early subdivision surface method for triangle or quadrilateral meshes (Fournier, Fussel, and Carpenter 1982). Later systems computed the maximum possible extent of the midpoint displacement surface of a mesh element and used this extent to construct a bounding volume for the mesh element (Kajiya 1983; Bouville 1985). This supported the lazy evaluation of the procedural terrain model. If the renderer determines that the bounding volume of a mesh element is not visible, then that element need not be subdivided and displaced.

Team LRN

Bounding volumes for procedural geometry can be computed statically or dynamically. A *static* bounding volume encases all possible geometries synthesized by the procedure, whereas a *dynamic* bounding volume is designed to tightly bound the specific output of a particular evaluation of the procedure.

Dynamic bounding volumes are more efficient than static bounding volumes, but they are also much more difficult to devise. Dynamic bounding volumes need to be computed at the time of instantiation. A bounding volume procedure is executed that predicts the extent of the procedural geometry based on the parameters used to synthesize the geometry.

Example: *Bounding Grass*

The bounding box hierarchy for the field of grass specified previously in an example is constructed by the dynamic bounding volume

$$\texttt{bounds}((0,0,0),2^{n+1}\,(.1,0,.1) + (0,0.3,0))$$

where `bounds` specifies a bounding box with two of its opposing corners. Each blade of grass is assumed to be 0.1 units thick and 0.3 units high.

Example: *Bounding Trees*

Under standard geometric instancing, a tree may be specified as a trunk and several smaller instances of the tree. Repeated forever, this generates a linear fractal, and the instancing structure is similar to an iterated function system (Hart 1992). Given an iterated function system, the problem of finding an optimal bounding volume for its attractor remains open, although several working solutions have appeared (Hart and DeFanti 1991; Canright 1994; Dubuc and Hamzaoui 1994).

Under procedural geometric instancing, tree subdivision gains additional freedom for global influences. Beginning from the trunk, it is a matter of chaos where each leaf is finally instanced. Bounding volumes are derived from prediction, which is difficult in such chaotic cases. Hence, we look to the worst possible case to obtain an efficient parametric bound on the geometry of a tree.

Trees, such as those described previously in an example, scale geometrically, with each branch a smaller replica of its parent. Geometric series, such as the length of branches from trunk to leaf, sum according to the formulas (and PGI built-in functions)

Team LRN

$$\sum_{i=0}^{\infty} x^i = \mathrm{sumx}(x) = \frac{1}{1-x} \tag{11.8}$$

$$\sum_{i=m}^{n} x^i = sumxmn(x, m, n) = \frac{x^m - x^n}{1-x} \tag{11.9}$$

Consider the definition of a tree whose branches scale by factors of λ_i. Let $\lambda = \max \lambda_i$ be the scaling factor of the major branch; for sympodial trees (Prusinkiewicz and Lindenmayer 1990), this is the extension of the trunk. Then from the base of the trunk, which is canonically set to unit length, the branches can extend no farther than $\mathrm{sumx}(\lambda)$ units away. When limited to n iterations, the branches are bound by a sphere of radius $\mathrm{sumxmn}(\lambda, 0, n)$ centered at the base of the trunk. At iteration m, these branches are bound by a sphere of radius $\mathrm{sumxmn}(\lambda, m, n)$ centered at the current point. While these bounds are quite loose initially, they converge to tighter bounds at higher levels.

While tight bounding volumes of near-terminal limbs will efficiently cull the rendering of hidden branches, a tight bound on the entire tree is also useful to avoid rendering trees that are hidden behind dense forests or mountain ridges. A simple worst-case analysis can precompute the maximum height a tree attains as well as the maximum width. These computations are less useful for the branches because the global influences may be different on the arbitrarily positioned and oriented instance of the tree. However, clipping the bounding volumes of the branches to the tree's bounding volume makes the branch bounding volumes tighter and more efficient.

CONCLUSION

Procedural geometric instancing is a language for articulating geometric detail. Its main impact is the insertion of procedural hooks into the scene graph. These hooks allow the scene graph to serve as a model for procedural geometry and allow it to overcome the intermediate storage problem. The resulting scene graph provides on-demand lazy evaluation of its instances performed at the time of instantiation, and only for objects affecting the current rendering of the scene.

Procedural geometric instancing is a geometric complement to shading languages. It provides the renderer with a procedural interface to the model's geometry definition in the same way that a shading language provides the renderer with a procedural interface to the model's shading definition.

Team LRN

Procedural geometric instancing yields geometries that are processed more efficiently than those generated by turtle graphics. Procedural geometric instancing is also based on the scene graph and its associated scene description, which is a more familiar and readable format for the articulation of procedural models than is an L-system's productions of turtle graphics symbols.

The parameterization and other features of procedural geometric instancing make standard textual geometric descriptions of natural models more compact and readable.

Procedural Geometric Modeling and the Web

Procedural modeling can play a critical role in multimedia, networking, and the World Wide Web. Two standards have recently become popular: Java and VRML. Java is a system that allows programs to be automatically loaded from a remote site and run safely on any architecture and operating system. The Virtual Reality Modeling Language (VRML) has become a standard for transmitting geometric scene databases.

In their current form, VRML geometric databases are transmitted in their entirety over the network, and Java has little support for generating complicated geometric databases. However, both can be enhanced to support the lazy evaluation paradigm for procedural modeling.

One example extends the capabilities of Java or an equivalent language to support the generation and hierarchical organization of detailed geometry. A user may download this script, and a renderer then runs it to procedurally generate the necessary geometry to accommodate the vantage point of the viewer. This example places the network at the "articulation" step of the paradigm.

A second example places the network at the "geometry/coordinates" bidirectional step of the lazy evaluation paradigm. In this example, a powerful server generates the geometry needed by a remote client renderer to view a scene and transmits only this geometry over the network. As the client changes viewpoint, the server then generates and transmits only the new geometry needed for the new scene.

Future Work

The major obstacle to efficient rendering for procedural geometric instancing is the construction of effective bounding volume hierarchies. Since geometry is created on demand, a bounding volume must be able to predict the extent of its contents. Such procedural bounding volumes were constructed for fractal terrain models by Kajiya

Team LRN

(1983) and Bouville (1985), but their generalization to arbitrary subdivision processes remains unsolved.

Amburn, Grant, and Whitted (1986) developed a system in which context was weighted between independent subdivision-based models. Fowler, Prusinkiewicz, and Battjes (1992) developed a geometric context-sensitive model of phyllotaxis based on the currently generated geometry of a procedural model. A similar technique could model the upward tropism of the tips of branches on some evergreen trees. These tips bend upward depending on the visibility of sunlight. If the tree was instanced from the highest branches, working its way down to the ground level, the visibility of each branch with respect to the previously instanced branches could be efficently computed using a separate frame buffer as a progressively updated "light" map.

ACKNOWLEDGMENTS

Procedural geometric instancing has been in various forms of development for nearly a decade, and some of this development was funded by Intel. Anand Ramagapalrao implemented the front-to-back sorting of hierarchical bounding volumes. Chanikarn Kulratanayan and Hui Fang also used the system for their research and generated numerous impressive example images. A significant portion of the ideas behind PGI resulted from conversations with Przemyslaw Prusinkiewicz, most occurring on various ski lifts across the Northwest.

Team LRN

12

Team LRN

NOISE, HYPERTEXTURE, ANTIALIASING, AND GESTURE

KEN PERLIN

INTRODUCTION

This first section touches on several topics that relate to the work I've done in procedural modeling. A brief introduction to hypertexture and a review of its fundamental concepts is followed by a discussion of the essential functions needed in order to be able to easily tweak procedural models.

The second section is a short tutorial on how the *noise* function is constructed, followed by a discussion of how raymarching for hypertexture works and some examples of hypertexture. Next comes an interesting possible approach for antialiasing procedurally defined images. Then we discuss a surface representation based on sparse wavelets. We conclude by applying notions from procedural modeling to human figure motion.

Shape, Solid Texture, and Hypertexture

The precursor to hypertexture was *solid texture*. Solid texturing is simply the process of evaluating a function over R^3 at each visible surface point of a rendered computer graphic (CG) model. The function over R^3 then becomes a sort of solid material, out of which the CG model shape appears to be "carved."

I became interested in what happens when you start extending these texture functions off of the CG surface. What do they look like as space-filling functions? So I developed a simple formalism that allowed me to play with this idea, by extending the notion of *characteristic function*.

Traditionally, the shape of any object in a computer graphic simulation is described by a characteristic function—a mapping from each point in R^3 to the

Team LRN

Boolean values **true** (for points inside the object) or **false** (for points outside the object). This is a point set. The boundary of this point set is the surface of the object.

I replaced the Boolean characteristic function with a continuous one. We define for any object a characteristic function that is a mapping from f: $R^3 \rightarrow [0 \ldots 1]$. All points $\vec{x}$ for which $f(\vec{x})$ is 0 are said to be *outside* the object. All points $\vec{x}$ for which $f(\vec{x})$ is 1 are said to be *strictly inside* the object. Finally, all points $\vec{x}$ for which $0 < f(\vec{x}) < 1$ are said to be in the object's *fuzzy region*.

This formulation gives the object surface an appreciable thickness. We can now combine solid texture functions with the function that describes the object's fuzzy region. In this way shape and solid texture become unified—a shape can be seen as just a particular solid texture function. I refer to this flavor of texture modeling as *hypertexture*, since we are texturing in a higher dimension (the full three-dimensional object space)—and also because it's a really atrocious pun.

TWO BASIC PARADIGMS

There are two distinct ways that solid textures can be combined with fuzzy shapes. You can either use the texture to distort the space before evaluating the shape, or add the texture value to a fairly soft shape function, then apply a "sharpening" function to the result.

The first approach involves calculations of the form

$$f(\vec{x}) = \text{shape}(\vec{x} + \text{texture}(\vec{x})\vec{v})$$

where $\vec{v}$ is a simple vector expression. The second approach involves calculations of the form

$$f(\vec{x}) = \text{sharpen}(\textit{shape}(\vec{x}) + \text{texture}(\vec{x}))$$

Bias, Gain, and So Forth

The secret to making good procedural textures is an interface that allows you to tune things in a way that makes sense to you. For example, let's say that there is some region over which a function value is varying from 0.0 to 1.0. Perhaps you realize that this function should be "pushed up" so that it takes on higher values in its middle range. Or maybe you want to push its values a bit toward its high and low values and away from the middle.

Team LRN

You could do these things with spline functions or with power functions. For example, let's say that $f(t)$: $R \rightarrow [0 \ldots 1]$. Then $pow(f(t), 0.8)$ will push the values of f up toward 1.0. But this is not intuitive to use because it's hard to figure out what values to put into the second argument to produce a particular visual result. For this reason, I've built two functions that I use constantly, just to do these little continuous tweaks.

Bias

To get the same functionality provided by the power function, but with a more intuitive interface, I've defined the function $bias_b$, which is a power curve defined over the unit interval such that $bias_b\,(0.0) = 0.0$, $bias_b\,(0.5) = b$, and $bias_b\,(1.0) = 1.0$. By increasing or decreasing b, we can thus bias the values in an object's fuzzy region up or down. Note that $bias_{0.5}$ is the identity function.

Bias is defined by

$$t^{\frac{\ln(b)}{\ln(0.5)}}$$

Gain

Similarly, we often want an intuitive way to control whether a function spends most of its time near its middle range or, conversely, near its extremes. It's sort of like having a gain on your TV set. You can force all the intensities toward middle gray (low gain) or, conversely, force a rapid transition from black to white (high gain).

We want to define $gain_g$ over the unit interval such that

$$\begin{aligned} gain_g\,(0.0) &= 0.0 \\ gain_g\,(0.25) &= 0.5 - g/2 \\ gain_g\,(0.5) &= 0.5 \\ gain_g\,(0.75) &= 0.5 + g/2 \\ gain_g\,(1.0) &= 1.0 \end{aligned}$$

By increasing or decreasing g, we can thus increase or decrease the rate at which the midrange of an object's fuzzy region goes from 0.0 to 1.0. Note that $gain_{0.5}$ is the identity function.

Team LRN

Motivated by the above, I've defined gain by

$$\text{if } t > 0.5 \text{ then } \frac{bias_{1-g}(2t)}{2} \text{ else } \frac{2 - bias_{1-g}(2 - 2t)}{2}$$

The above two functions provide a surprisingly complete set of tools for tweaking things. If you want to make something look "darker" or "more transparent," you would generally use $bias_b$ with $b < 0.5$. Conversely, if you want to make something look "brighter" or "more opaque," you would generally use $bias_b$ with $b > 0.5$.

Similarly, if you want to "fuzz out" a texture, you would generally use $gain_g$ with $g < 0.5$. Conversely, if you want to "sharpen" a texture, you would generally use $gain_g$ with $g > 0.5$. Most of the time, you'll want to instrument various numerical parameters with bias and gain settings and tune things to taste.

CONSTRUCTING THE *NOISE* FUNCTION

Part of the reason that procedural texture is effective is that it incorporates randomness in a controlled way. Most of the work involved in achieving this is contained inside the *noise* function, which is an approximation to what you would get if you took white noise (say, every point in some sampling mapped to a random value between −1.0 and 1.0) and blurred it to dampen out frequencies beyond some cutoff.

The key observation here is that, even though you are creating "random" things, you can use *noise* to introduce high frequencies in a controlled way by applying *noise* to an appropriately scaled domain. For example, if $\vec{x}$ is a point in R^3, then *noise* $(2\vec{x})$ introduces frequencies twice as high as does *noise* $(\vec{x})$. Another way of saying this is that it introduces details that are twice as small.

Ideally the *noise* function would be constructed by blurring white noise, preferably by convolving with some Gaussian kernel. Unfortunately, this approach would necessitate building a volume of white noise and then blurring it all at once. This is quite impractical.

Instead, we want an approximation that can be evaluated at arbitrary points, without having to precompute anything over some big chunk of volume. The approach described here is the original one I came up with in 1983, which I still use. It's a kind of spline function with pseudorandom knots.

The key to understanding the algorithm is to think of space as being divided into a regular lattice of cubical cels, with one pseudorandom wavelet per lattice point.

Team LRN

You shouldn't get scared off by the use of the term "wavelet." A wavelet is simply a function that drops off to zero outside of some region and that integrates to zero. This latter condition means that the wavelet function has some positive region and some negative region, and that the two regions balance each other out. When I talk about the "radius" of a wavelet, I just mean the distance in its domain space from the wavelet center to where its value drops off to zero.

We will use wavelets that have a radius of one cel. Therefore, any given point in R^3 will be influenced by eight overlapping wavelets—one for each corner of the cel containing the point.

Computation proceeds in three successive steps:

- Compute which cubical "cel" we're in.
- Compute the wavelet centered on each of eight vertices.
- Sum the wavelets.

Computing Which Cubical "Cel" We're In

The "lowest" corner of the cel containing point $[x, y, z]$ is given by

$$[i, j, k] = [\lfloor x \rfloor, \lfloor y \rfloor, \lfloor z \rfloor]$$

Then the eight vertices of the cel containing $[x, y, z]$ are

$$\begin{aligned}
&(1)\ [i, j, k] \\
&(2)\ [i + 1, j, k] \\
&(3)\ [i, j + 1, k] \\
&\quad \ldots \\
&(8)\ [i + 1, j + 1, k + 1]
\end{aligned}$$

Finding the Pseudorandom Wavelet at Each Vertex of the Cel

For each of the above lattice points, we want to find what wavelet to apply. Ideally, the mapping from cels to wavelet coefficients would be nonrepeating. It turns out, though, that because *noise* has no large-scale structure, the mapping can eventually repeat itself, without this repetition being at all noticeable. This allows us to use a relatively simple table lookup scheme to map vertices to coefficients. I find that a repeat distance of 256 is more than large enough.

Team LRN

The basic idea is to map any $[i, j, k]$ into a unique number between 0 and 255. We precompute a table G of 256 pseudorandom coefficient values and always index into this fixed set of values.

In the following two sections we will accomplish two things. First, we will discuss what properties we want from the wavelet coefficients and how to precompute G—a table of 256 sets of coefficients that has these properties. Then we will discuss the properties we want in the mapping from $P : [i, j, k] \rightarrow 0 \ldots 255$ and how to achieve these properties.

Wavelet Coefficients

I actually force the *noise* function to take on a value of zero at each lattice point. This ensures that it can't have any appreciable energy at low spatial frequencies. To achieve this, I give a value at its center of zero to each of the overlapping wavelets that are being summed and then give each wavelet a radius of one—so that each wavelet will reach a value of zero just as it gets to the center of its neighbors.

This means that we can define the wavelet centered at each lattice point as a smooth function with the following properties:

- It has a value of zero at its center.
- It has some randomly chosen gradient at its center.
- It smoothly drops off to zero a unit distance from its center.

So we have to randomly choose a gradient for each wavelet center. I do this via a Monte Carlo method—precompute a table of gradients in uniformly random directions and then index into this table. To create the gradient table, I need a set of points that are uniformly distributed on the surface of a unit sphere. I ensure that this set of points will be uniformly distributed as follows:

- Choose points uniformly within the cube $[-1 \ldots 1]^3$.
- Throw out any points falling outside of the unit sphere.
- Project surviving points onto the unit sphere.

The pseudocode to do this is

```
for i in [0 . . . 255]
  repeat
    x = random(−1. ··· +1.)
```

$y = random(-1. \cdots +1.)$
$z = random(-1. \cdots +1.)$
until $x^2 + y^2 + z^2 < 1.0$
$G[i] = normalize\ [x, y, z]$

To Quickly Index into *G* in a Nonbiased Way

Now we need to find a way to answer the following question: If any point in R^3 is in the vicinity of a wavelet, how can it rapidly get the coefficients of that wavelet?

We basically want a really random way to map lattice points $[i, j, k]$ to indices of G. We have to avoid regularities in this mapping, since any regularity would be extremely visible to somebody looking at the final *noise* function.

I use the following method:

- Precompute a "random" permutation table P.
- Use this table to "fold" $[i, j, k]$ into a single n.

In this section I describe first how to create a suitable permutation table and then how to use this table to "fold" $[i, j, k]$ into the range $[0 \ldots 255]$.

The permutation table can be precomputed, so that step doesn't need to be especially fast. The pseudorandom permutation table P is created by

for i in $[0 \ldots 255]$
  $j = random[0 \ldots 255]$
  exchange $P[i]$ *with* $P[j]$

The folding function $fold(i, j, k)$ is then computed by

$$n = P[i_{\text{mod } 256}]$$
$$n = P[(n + j)_{\text{mod } 256}]$$
$$n = P[(n + k)_{\text{mod } 256}]$$

For added speed, I don't actually do the mod operations. Instead, I precompute P to be twice as long, setting $P[256 \ldots 511] := P[0 \ldots 255]$. Then if $0 \le i, j, k \le 255$, we can just do $P[P[P[i] + j] + k]$ for each wavelet.

Now for each vertex of the unit cube whose "lowest" corner is $[i, j, k]$, we can quickly obtain wavelet coefficients as follows:

(1) $G(fold(i, j, k))$
(2) $G(fold(i + 1, j, k))$

$$(3)\ G(fold(i, j + 1, k))$$
$$\ldots$$
$$(8)\ G(fold(i + 1, j + 1, k + 1))$$

Evaluating the Wavelet Centered at [i, j, k]

The remaining steps to finding $noise(x, y, z)$ are now as follows:

1. Each wavelet is a product of
   - a cubic weight that drops to zero at radius 1
   - a linear function, which is zero at (i, j, k)
2. To compute the wavelet we must
   - get (x, y, z) relative to the wavelet center
   - compute the weight function
   - multiply the weight by the linear function

   First we must get (x, y, z) relative to the wavelet center:

$$[u, v, w] = [x - i, y - j, z - k]$$

Note that u, v, w values are bounded by $-1 \le u, v, w \le 1$.

Now we must compute the dropoff $\Omega_{(i,j,k)}(u, v, w)$ about $[i, j, k]$:

$$\Omega_{(i,j,k)}(u, v, w) = drop(u) \times drop(v) \times drop(w)$$

where each component dropoff is given by the cubic approximation

$$\text{drop}(t) = 1. - 3|t|^2 + 2|t|^3 \qquad \text{(but zero whenever } |t| > 1)$$

and we must multiply this by the linear function that has the desired gradient and a value of zero at the wavelet center:

$$G_{(i,j,k,)} \cdot [u, v, w]$$

The value of $\text{wavelet}_{(i,j,k)}$ at (x, y, z) is now given by

$$\Omega_{(i,j,k)}(u, v, w)(G_{(i,j,k)} \cdot [u, v, w])$$

Finally, $noise\ (x, y, z)$ is given by the sum of the eight wavelets near (x, y, z).

Following is a complete implementation of *noise* over R^3:

```
/* noise function over R3-implemented by a pseudorandom tricubic spline */

#include <stdio.h>
#include <math.h>
```

```
#define DOT(a,b) (a[0] * b[0] + a[1] * b[1] + a[2] * b[2])

#define B 256

static p[B + B + 2];
static float g[B + B + 2][3];
static start = 1;

#define setup(i,b0,b1,r0,r1) \
     t = vec[i] + 10000.; \
     \ b0 = ((int)t) & (B-1); \
     b1 = (b0+1) & (B-1); \
     r0 = t - (int)t; \
     r1 = r0 - 1.;

float noise3(vec)
float vec[3];
{
     int bx0, bx1, by0, by1, bz0, bz1, b00, b10, b01, b11;
     float rx0, rx1, ry0, ry1, rz0, rz1, *q, sy, sz, a, b, c, d, t, u, v;
     register i, j;

     if (start) {
          start = 0;
          init();
     }

     setup(0, bx0,bx1, rx0,rx1);
     setup(1, by0,by1, ry0,ry1);
     setup(2, bz0,bz1, rz0,rz1);

     i = p[ bx0 ];
     j = p[ bx1 ];

     b00 = p[ i + by0 ];
     b10 = p[ j + by0 ];
     b01 = p[ i + by1 ];
     b11 = p[ j + by1 ];

#define at(rx,ry,rz) ( rx * q[0] + ry * q[1] + rz * q[2] )

#define s_curve(t) ( t * t * (3. - 2. * t) )

#define lerp(t, a, b) ( a + t * (b - a) )

     sx = s_curve(rx0);
     sy = s_curve(ry0);
     sz = s_curve(rz0);
```

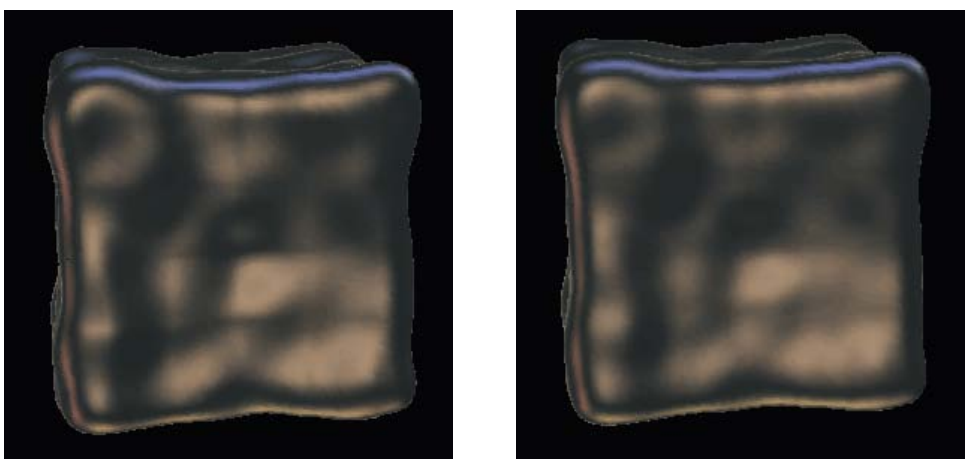
Team LRN

```
     q = g[ b00 + bz0 ] ; u = at(rx0,ry0,rz0);
     q = g[ b10 + bz0 ] ; v = at(rx1,ry0,rz0);
     a = lerp(sx, u, v);

     q = g[ b01 + bz0 ] ; u = at(rx0,ry1,rz0);
     q = g[ b11 + bz0 ] ; v = at(rx1,ry1,rz0);
     b = lerp(sx, u, v);

     c = lerp(sy, a, b);          /* interpolate in y at low x */

     q = g[ b00 + bz1 ] ; u = at(rx0,ry0,rz1);
     q = g[ b10 + bz1 ] ; v = at(rx1,ry0,rz1);
     a = lerp(sx, u, v);

     q = g[ b01 + bz1 ] ; u = at(rx0,ry1,rz1);
     q = g[ b11 + bz1 ] ; v = at(rx1,ry1,rz1);
     b = lerp(sx, u, v);

     d = lerp(sy, a, b);          /* interpolate in y at high x */

     return 1.5 * lerp(sz, c, d); /* interpolate in z */
}

static init()
{
     long random();
     int i, j, k;
     float v[3], s;

     /* Create an array of random gradient vectors uniformly on the
     unit sphere */
     srandom(1);
     for (i = 0 ; i < B ; i++) {
          do {                    /* Choose uniformly in a cube */
               for (j=0 ; j<3 ; j++)
                    v[j] = (float)((random() % (B + B)) - B)/B;
                    s = DOT(v,v);
          } while (s > 1.0);      /* If not in sphere try again */
          s = sqrt(s);
          for (j = 0 ; j < 3 ; j++)      /* Else normalize */
               g[i][j] = v[j] / s;
     }

     /* Create a pseudorandom permutation of [1 .. B] */
     for (i = 0 ; i < B ; i++)
          p[i] = i;
     for (i = B ; i > 0 ; i -= 2) {
          k = p[i];
          p[i] = p[j = random() % B];
          p[j] = k;
```

```
    }

    /* Extend g and p arrays to allow for faster indexing, */
    for (i = 0 ; i < B + 2 ; i++) {
        p[B + i] = p[i];
        for (j = 0 ; j < 3 ; j++)
        g[B + i][j] = g[i][j];
    }
}
```

RECENT IMPROVEMENTS TO THE *NOISE* FUNCTION

I recently made some tweaks to the *noise* function that make it look better and run somewhat faster (Perlin 2002). Note that the cubic interpolation polynomial $3t^2 - 2t^3$ has a derivative of $6 - 12t$, which has nonzero first and second derivatives at $t = 0$ and $t = 1$. These create second-order discontinuities across the coordinate-aligned faces of adjoining cubic cells, which become noticeable when a noise-displaced surface is shaded, and the surface normal (which is itself a derivative operator) acquires a visibly discontinuous derivative. We can instead use the interpolation polynomial $6t^5 - 15t^4 + 10t^3$, which has zero first and second derivatives at $t = 0$ and $t = 1$. The improvement can be seen by comparing the two noise-displaced superquadrics (see Figure 12.1).



(a) (b)

FIGURE 12.1 (a) Noise interpolated with $3t^2 - 2t^3$; (b) noise interpolated with $6t^5 - 15t^4 + 10t^3$.

Also, we can speed up the algorithm while reducing grid-oriented artifacts, by replacing the *g* array with a small set of fixed gradient directions:

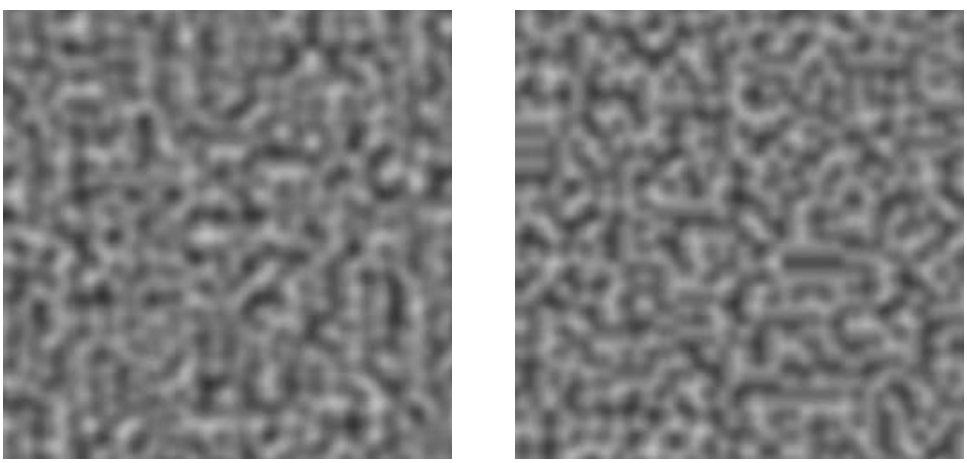
$$(1, 1, 0),(-1, 1, 0),(1, -1, 0),(-1, -1, 0),$$
$$(1, 0, 1),(-1, 0, 1),(1, 0, -1),(-1, 0, -1),$$
$$(0, 1, 1),(0, -1, 1),(0, 1, -1),(0, -1, -1)$$

This set of gradient directions does two things: (1) it avoids the main axis and long diagonal directions, thereby avoiding the possibility of axis-aligned clumping, and (2) it allows the eight inner products to be effected without requiring any multiplies, thereby removing 24 multiplies from the computation. The improvement can be seen by comparing the two gradient strategies (see Figure 12.2).

RAYMARCHING

To see hypertexture, you need a *raymarcher* renderer. I've implemented the raymarcher in two parts. First I built a system layer—the part that never changes. This renders hypertexture by marching a step at a time along each ray, accumulating density at each sample along the way. There are all kinds of hooks for user interaction built into the raymarcher.

Then, for each type of hypertexture, I construct different "application" code, describing a particular hypertexture. At this point in the process I can safely wear the



(a) (b)

FIGURE 12.2 (a) Noise with old gradient distribution; (b) noise with new gradient distribution.

Team LRN

hat of a "naive user." Everything is structured so that at this level I don't have to worry about any system details. All I really need to worry about is what density to return at any point in R^3. This makes it very painless to try out different types of hypertexture.

System Code: The Raymarcher

I render hypertexture by "raymarching"—stepping front to back along each ray from the eye until either total opacity is reached or the ray hits a back clipping plane. Conceptually, the raymarching is done inside the unit cube: $-0.5 < x, y, z < 0.5$. A 4×4 viewing matrix transforms this cube to the desired view volume.

One ray is fired per pixel; the general procedure is as follows:

```
step = 1.0/resolution;
for (y = -0.5 ; y < 0.5 ; y += step)
for (x = -0.5 ; x < 0.5 ; x += step) {
    [point, point_step] = create_ray([x,y,-0.5], [x,y,-0.5+step], view_matrix);
    previous_density = 0.;
    init_density_function();                /* User supplied */
    color = [0,0,0,0];
    for (z = -0.5 ; z < 0.5 && color.alpha < 0.999 ; z += step} {

         density = density_function(point); /* User supplied */
         c = compute_color(density);        /* User supplied */

    /* Do shading only if needed */

        if (is_shaded && density != previous_density) {
            normal = compute_normal(point, density);
            c = compute_shading(c, point, normal); /* User supplied */
            previous_density = density;
        }

    /* Attenuation varies with resolution */

        c[3] = 1.0 - pow( 1.0 - c[3], 100. * step );

    /* Integrate front to back */

        if (c[3] > 0.)} {
            t = c[3] * (1.0 - color.alpha);
            color += [ t*c.red, t*c.green, t*c.blue, t ];
             }
```

```
    /* March further along the ray */

       point += point_step;
    }
}
```

Application Code: User-Defined Functions

The "user" gets to define four functions to create a particular hypertexture:

```
void init_density_function();
float density_function(float x, float y, float z);
color compute_color(float density);
color compute_shading(color c, vector point, vector normal);
```

What makes things really simple is that as a user you only have to define behavior at any given single point in space—the raymarcher then does the rest for you.

- `init_density_function()`—This function is called once per ray. It gives you a convenient place to compute things that don't change at every sample.
- `density_function()`—This is where you specify the mapping from points to densities. Most of the behavior of the hypertexture is contained in this function.
- `compute_color()`—Here you map densities to colors. This also gives you a chance to calculate a refractive index.
- `compute_shading()`—Nonluminous hypertextures react to light and must be shaded. The model I use is to treat any substance that has a density gradient as a translucent surface, with the gradient direction acting as a normal vector, as though the substance consists of small, shiny, suspended spheres.

In the raymarcher library I've included a Phong shading routine. I usually just call that with the desired light direction, highlight power, and so on.

Shading is relatively expensive, since it requires a normal calculation. Also, in many cases (e.g., self-luminous gases) shading is not necessary. For this reason, shading is only done if the user sets an `is_shaded` flag.

The raymarcher computes normals for shading by calling the user's density function three extra times:

```
vector = compute_normal(point, density) {
   vector d = = [
       density_function[point.x - epsilon, point.y, point.z] - density,
       density_function[point.x, point.y - epsilon, point.z] - density,
       density_function[point.x, point.y, point.z - epsilon] - density ];
   return d / |d|;
}
```

The preceding is the basic raymarcher. Two features have not been shown—refraction and shadows. Shadows are done by shooting secondary rays at each ray step where density != 0. They are prohibitively expensive except for hypertextures with "hard," fairly sharpened surfaces. In this case the accumulated opacity reaches totality in only a few steps, and so relatively few shadow rays need be followed.

Refraction is done by adding a fifth component to the color vector—an index of refraction. The user sets `c.irefract` (usually from density) in the `compute_color` function. The raymarcher then uses Snell's law to shift the direction of `point_step` whenever `c.irefract` changes from one step along the ray to the next. An example of this is shown in Figure 12.3.

Since the density can change from one sample point to the next, it follows that the normal vector can also change continuously. This means that refraction can occur continuously. In other words, light can travel in curved paths inside a

FIGURE 12.3 Blue glass.

hypertexture. This raises some interesting possibilities. For example, imagine a manufacturing process that creates wafers whose index of refraction varies linearly from one face to the other (probably by some diffusion process). By carving such a material, you could create optical components within which light travels in curved paths. It might be possible to do things this way that would be very difficult or impossible to do with traditional optical components (in which light only bends at discrete surfaces between materials). The results of such components should be quite straightforward to visualize using refractive hypertexture.

INTERACTION

Various kinds of interaction with hypertextures are possible, including modification of parameters and algorithmic models and various types of previewing, such as *z-slicing*.

Levels of Editing: Changing Algorithms to Tweaking Knobs

There are three levels of changes you can make. I describe them in order of slowest to implement (and most sweeping in effect) to fastest to implement.

- *Semantic changes—changing the user functions:* This is redefining your hypertexture methods. This type of change is covered in detail in the next section.
- *Parameters you change by editing an input file of numeric parameters:* This saves the time of recompiling the user functions, when all you want to change is some numeric value. The raymarcher has a mechanism built into it that lets you refer to a file that binds symbols to floating-point values when you run the program. These bindings are made accessible to the hypertexture designer at the inner rendering loop.

  In the following examples, I will adopt the following convention: Any symbol that begins with a capital letter refers to a parameter whose value has been set in this input file. Symbols beginning with lowercase letters refer to variables that are computed within the individual rays and samples.
- *Parameters you change from the command line:* These override any parameters with the same name in the input file. They are used to make animations showing things changing. For example, let's say you want to create an animation of a sphere with an expanding `Radius`, and you are working in the UNIX `csh` shell:

Team LRN

```
set i = 0
while ($i < 100)
   rm hypertexture -Radius $i sphere > $i
   @ i++
end
```

There are also some special parameters: `XFORM` for the view matrix, `RES` for image resolution, and `CLIP` for image clipping (when you just want to recalculate part of an image). These can be set either from the command line or as an environment variable (the former overrides the latter, of course).

In this chapter, I have hardwired numerical parameters into a number of expressions. These are just there to "tune" the model in various useful ways. For example, the expression "`100 * step`" appearing above in the attenuation step of the raymarcher has the effect of scaling the integrated density so that the user can get good results by specifying densities in the convenient range [0.0 . . . 1.0].

z-Slicing

For much of the time when designing hypertextures, you just need a general sense of the shape and position of the textured object. In this case it is useful to evaluate only at a fixed *z*—setting the value of a ray to the density at only one sample point a fixed distance away. This obviously runs many times faster than a full raymarch. I use z-slicing for general sizing and placement, often going through many fast iterations in z-slice mode to get those things just right.

SOME SIMPLE SHAPES TO PLAY WITH

Generally, the creation of a hypertexture begins with an algorithmically defined shape. This is the shape that will subsequently be modified to create the final highly detailed hypertexture.

Sphere

Start with a sphere with `inner_radius` and `outer_radius` defined. Inside `inner_radius`, density is everywhere 1.0. Beyond `outer_radius`, density has dropped completely to 0.0. The interesting part is the hollow shell in between:

```
/* (1) Precompute (only once) */

    rr0 = outer_radius * outer_radius;
    rr1 = inner_radius * inner_radius;
```

```
/* (2) radius squared */

    t = x * x + y * y + z * z;

/* (3) compute dropoff */

    if (t > rr0)
        return 0.;
    else if (t < rr1)
        return 1.;
    else
        return (t - rr0) / (rr1 - rr0);
```

Egg

To create an egg, you start with a sphere, but distort it by making it narrower at the top. A good maximal "narrowing" value is 2/3, which is obtained by inserting the following step into the sphere procedure:

```
/* (1.5) introduce eccentricity */

    e = ( 5. - y / outer_radius ) / 6.;
    x = x / e;
    z = z / e;
```

Notice that we must divide, not multiply, by the scale factor. This is because `x` and `z` are the arguments to a shape-defining function—to make the egg thinner at the top, we must increase (not decrease) the scale of `x` and `z`.

EXAMPLES OF HYPERTEXTURE

We now show various examples of hypertexture. Each example will illustrate one or more hypertexture design principles.

Explosions

The texture component here is turbulence uniformly positioned throughout space.

```
t = 0.5 + Ampl * turbulence(x, y, z);
return max(0., min(1. t));
```

Shape is just a sphere with `inner_radius` = 0.0, which ensures that the fuzzy region will consist of the entire sphere interior.

The density function is

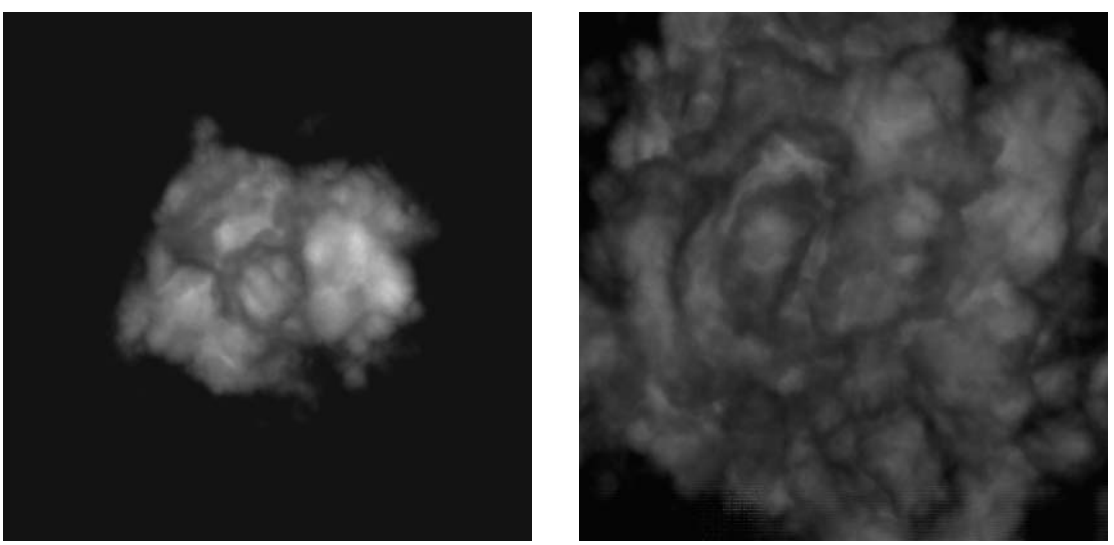
```
d = shape(x, y, z);
if (d > 0.)
    d = d * texture(x, y, z);
return d;
```

You can animate an explosion by increasing the sphere `outer_radius` over time. Figure 12.4(a) shows an explosion with `outer_radius` set to 0.4. Figure 12.4(b) shows the same explosion with `outer_radius` set to 0.8.

To create these explosions I oriented the cusps of the texture inward, creating the effect of locally expanding balls of flame on the surface. Contrast this with Figure 12.5 (Perlin and Hoffert 1989), where the cusps were oriented outward to create a licking flame effect.

Life-Forms

Just for fun, I placed a shape similar to the above explosions inside of an egg shape of constant density, as in Figure 12.6. By pulsing the `outer_radius` and `Ampl`



(a) (b)

FIGURE 12.4 (a) Explosion with `outer_radius` set to 0.4; (b) same explosion with `outer_radius` set to 0.8.

Team LRN

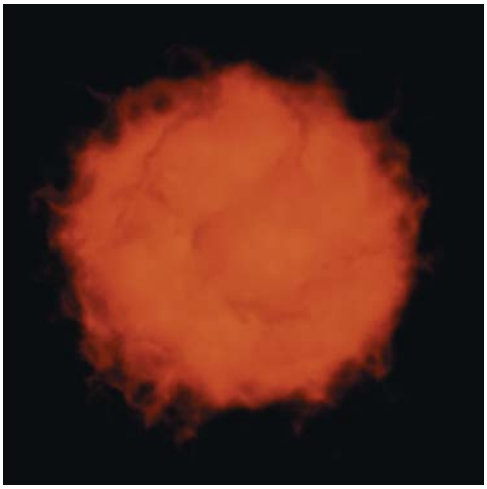


FIGURE 12.5 Fireball.

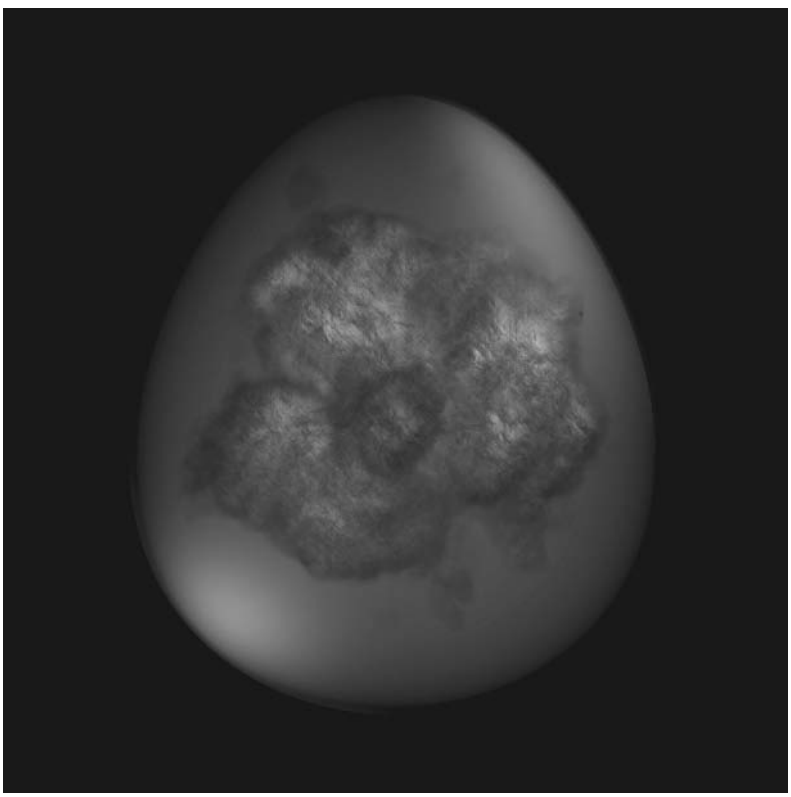


FIGURE 12.6 Explosion shape inside an egg.

Team LRN

rhythmically, while rotating slightly over time, I managed to hatch some rather intriguing simulations.

Space-Filling Fractals

Figure 12.7(a) through 12.7(d) shows steps in the simulation of a sparsely fractal material. At each step, *noise*() is used to carve volume away from the egg. Then *noise*() of twice the frequency is carved away from the remainder, and so on.

Figure 12.8 shows one possible result of such a process, a shape having infinite surface area and zero volume.

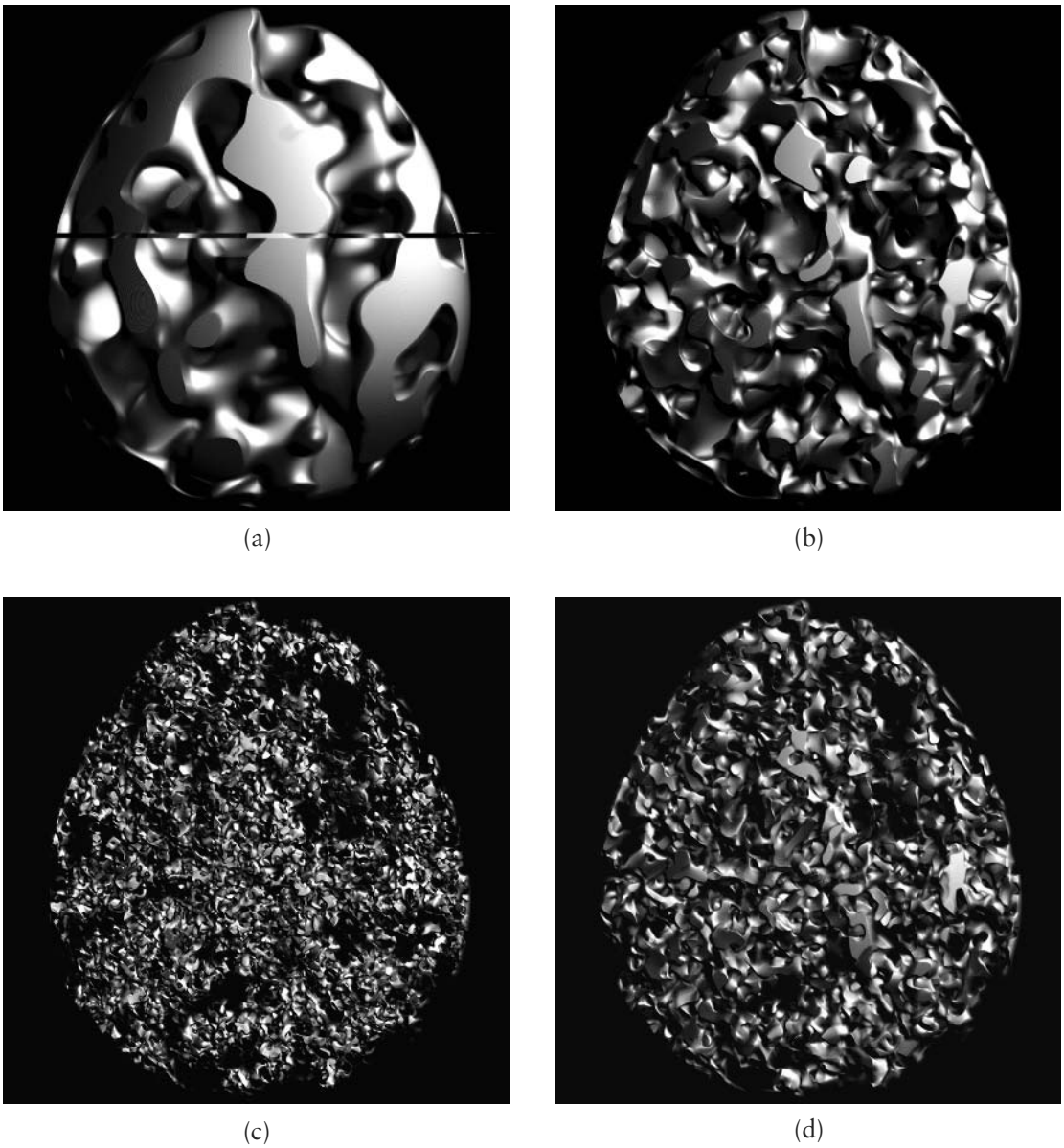
Woven Cloth

Cloth is defined by the perpendicular interweaving of warp threads and woof threads. We define a warp function `warp(x, y, z)`, where `y` is the direction perpendicular to the cloth:

```
/* (1) make an undulating slab */

   if (fabs(y) > PI)
       return 0.;

   y = y + PI/2 * cos(x) * cos(z);
   if (fabs(y) > PI/2)
       return 0.;

   density = cos(y);

/* (2) separate the undulating slab into fibers via cos(z) */

   density = density * cos(z);

/* (3) shape the boundary into a hard surface */

   density = density * density;
   density = bias(density, Bias);
   density = gain(density, Gain);
   return density;
```

We can then define a `woof` function by rotating 90 degrees in `z`, `x`, and flipping in `y`. The complete cloth function is then

```
    cloth(x, y, z) = warp(x, y, z) + warp(z, -y, x);
```



(a) (b)

(c) (d)

FIGURE 12.7 Steps in the simulation of a sparsely fractal material.

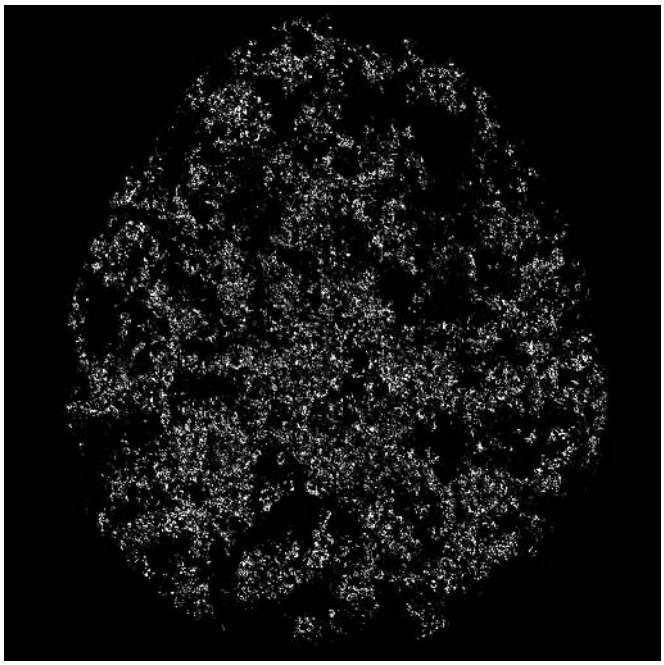


FIGURE 12.8 A shape having infinite surface and zero volume.

You can make the cloth wrinkle, fold, and so on, by transforming `x`, `y`, and `z` before applying the cloth function. You can also add high frequency *noise*() to `x`, `y`, `z` before applying `cloth()`, to simulate the appearance of roughly formed fibers. In the examples shown I have done both sorts of things.

In the cloth examples shown here, I "sharpen" the surface by applying the `bias()` and `gain()` functions. Figure 12.9(a) through 12.9(d) shows extreme close-ups of cloth with various bias and gain settings. Figure 12.9(a) has low bias and gain. In Figure 12.9(b) I increase gain, which "sharpens" the surface. In Figure 12.9(c) I increase bias, which expands the surface, in effect fattening the individual threads. In Figure 12.9(d) I increase both bias and gain. Figure 12.10 shows a high-resolution rendering of a low-bias, high-gain cloth, which gives a "thread" effect. Conversely, a high bias, low gain would give a "woolen" effect.

ARCHITEXTURE

Now let's take an architectural sketch and "grow" solid texture around it, ending up with hard textured surfaces. This is similar in spirit to Ned Greene's voxel automata

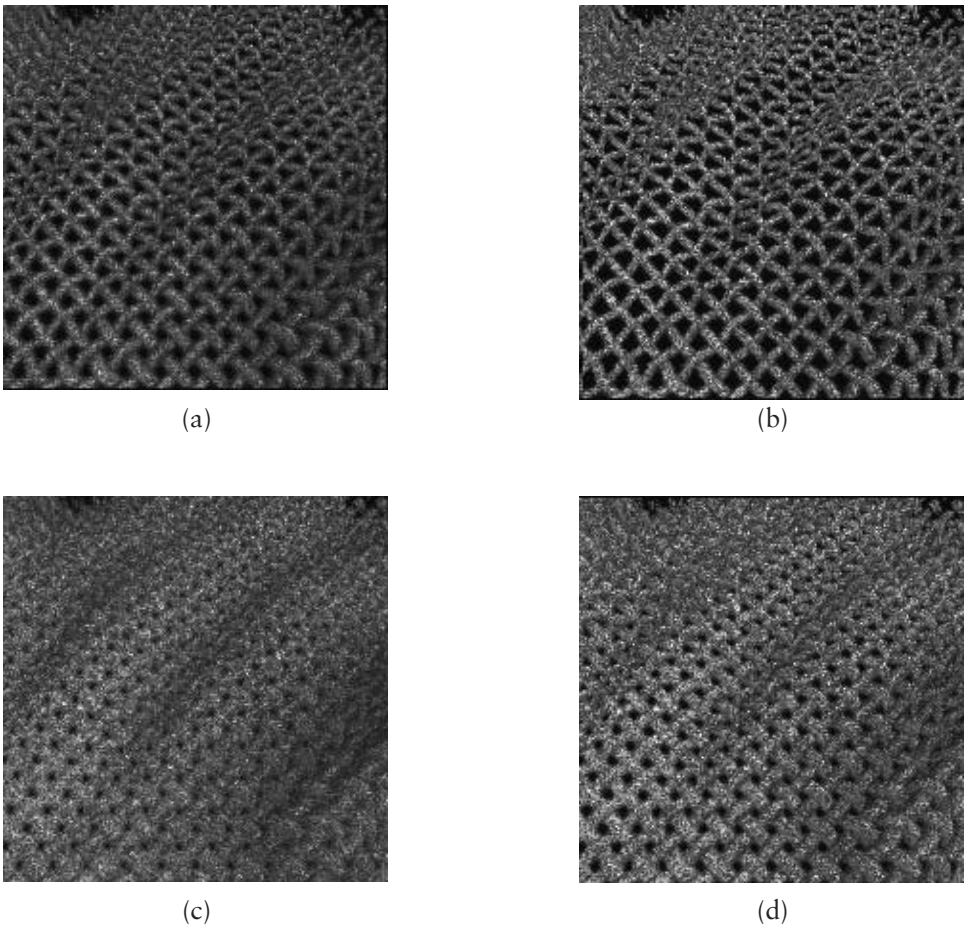


FIGURE 12.9 Extreme close-ups of cloth with various bias and gain settings.

algorithm. The difference is that whereas he literally "grows" a volume from a defining skeleton, one progressive voxel layer at a time, the hypertexture approach directly evaluates its result independently at each point in space.

I start with a skeleton of architectural elements. This can be supplied by freehand drawing or, alternatively, generated from a CAD program. Each architectural element is a "path" in space formed by consecutive points P_i.

Each path defines an influence region around it, which gives the architexture its shape component. This region is created by "blurring" the path. To do this I treat

Team LRN

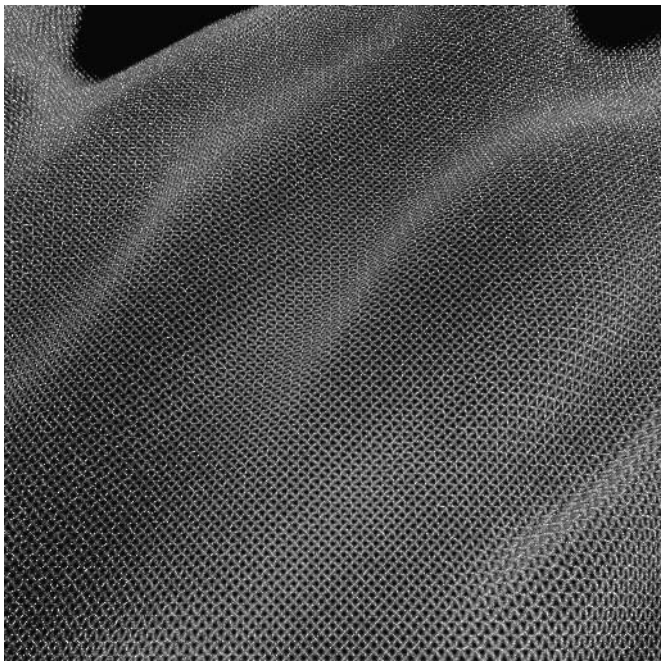


FIGURE 12.10 High-resolution rendering of a low-bias, high-gain cloth gives a "thread" effect.

each point along the path as the center of a low-density soft sphere of radius R. The shape density at a given point $\vec{x}$ is given by

$$\text{path\_shape}(\vec{x}) = \frac{1}{KR^2} \sum_i \max(0, R^2 - |P_i - \vec{x}|^2)$$

where the normalizing constant K is the distance between successive points on the path. For each volume sample, the cost per path point is a dot product and some adds, which is fairly expensive. To speed things up I maintain a bounding box around each path, which eliminates most paths from consideration for any given sample point.

I've only played so far with rocklike textures for architexture. The texture component of this is given by a simple noise-based fractal generator:

$$\text{rock\_texture}(\vec{x}) = \sum_{f=\log base\_freq}^{\log resolution} 2^{-f} noise(2^f \vec{x})$$

Team LRN

and I define the final density by

$$\text{sharpen}(\text{path\_shape}(\vec{x}) + \text{rock\_texture}(\vec{x}))$$

where I use the sharpening function to reduce the effective fuzzy region size about one volume sample. For a given image resolution and shape radius R, correct sharpen is done by

- scaling the density gradient about 0.5 by a factor of $1/R$ (adjusting also for variable image resolution)
- clipping the resulting density to between 0.0 and 1.0

The idea of the above is that the larger R becomes, the smaller will be the gradient of density within the fuzzy region, so the more sharpening is needed. The actual code I use to do this is

```
density = (density - 0.5) * (resolution / 600) / R + 0.5;
density = max(0.0, min(1.0, density));
```

I also decrease the shape's radius R with height (y-coordinate), which gives architectural elements a sense of being more massive in their lower, weight-supporting regions.

In Figures 12.11 and 12.12 I show a typical arch (which I suppose could archly be called archetypical architexture). This started out as a simple curved line tracing the inner skeleton, which was then processed as described above.

Figure 12.11 shows a sequence where three parameters are being varied. From left to right the width of the arch at the base increases. From top to bottom the thickness enhancement toward the base increases. These two parameters act in concert to add a "weight-supporting" character to architextural elements. Finally, the amplitude of the texture is increased linearly from the first image in the sequence to the last. Figure 12.12 shows a high-resolution version of the bottom-right image in the sequence.

The NYU Torch

In Figure 12.13, a number of hypertextures were used for various parts of the same object to build an impression of this well-known New York University icon. The various amounts of violet reflected down from the flame to the torch handle were just added manually, as a function of y.

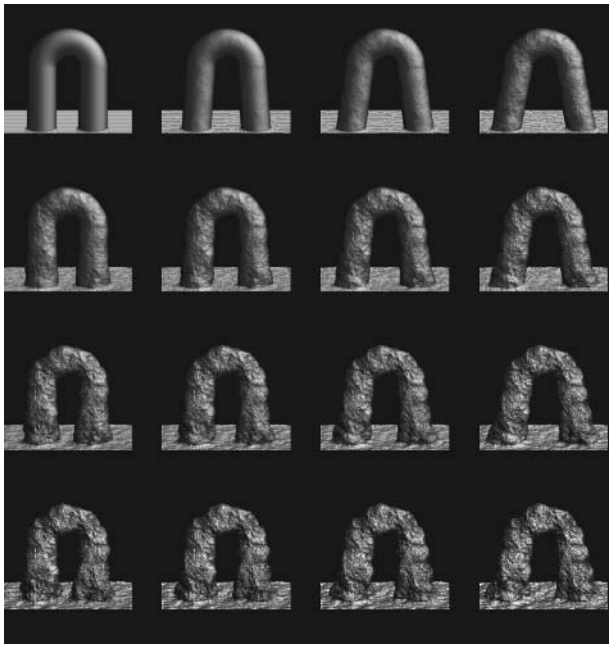


FIGURE 12.11 Arch with varying parameter values.

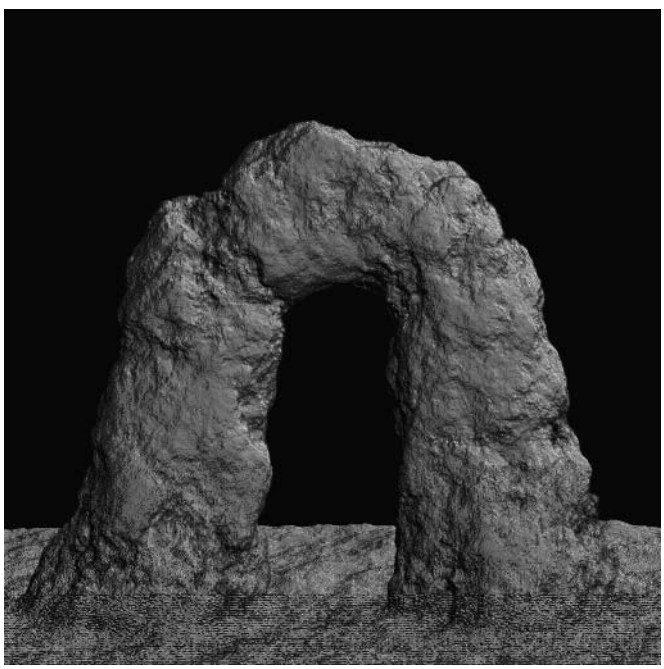


FIGURE 12.12 High-resolution version of the last image in the arch sequence.

Team LRN

FIGURE 12.13 New York University torch.

Smoke

In a recent experiment we tried to create the look of an animating smoke column, using as simple a hypertexture as possible. This work was done in collaboration with Ajay Rajkumar at NYU.

The basic approach was to create a smooth smoke "column" along the *y*-axis and then to perturb this column in *x*,*z*—increasing the perturbation at greater *y*

values. We added knobs for such things as column opacity and width. These "shaping" knobs can have a great effect on the final result. For example, Figure 12.14(a) and 12.14(b) vary only in their column width. Yet the difference in appearance between them is drastic.

Time Dependency

We make the smoke appear to "drift" in any direction over time by moving the domain of the turbulence in the *opposite* direction. In the following example, we do this domain shift in both x and y.

We move y linearly downward, to give the impression of a rising current. We move x to the left but increase the rate of movement at greater y values. This creates the impression that the smoke starts out moving vertically, but then drifts off to the right as it dissipates near the top.

The particular shape of the smoke can vary dramatically over time, yet the general *feel* of the smoke stays the same. Compare, for example, the two images in Figure 12.15(a) and 12.15(b), which are two frames from the same smoke animation.

Smoke Rings

Figure 12.16 shows the formation over time of a smoke ring. Smoke rings will occur when the turbulence function distorts the space sufficiently so that the column appears to double over on itself horizontally. We need to let this happen only fairly high up on the column. If it happens too low, then the rings will appear to be forming somewhere off in space, not out of the column itself.

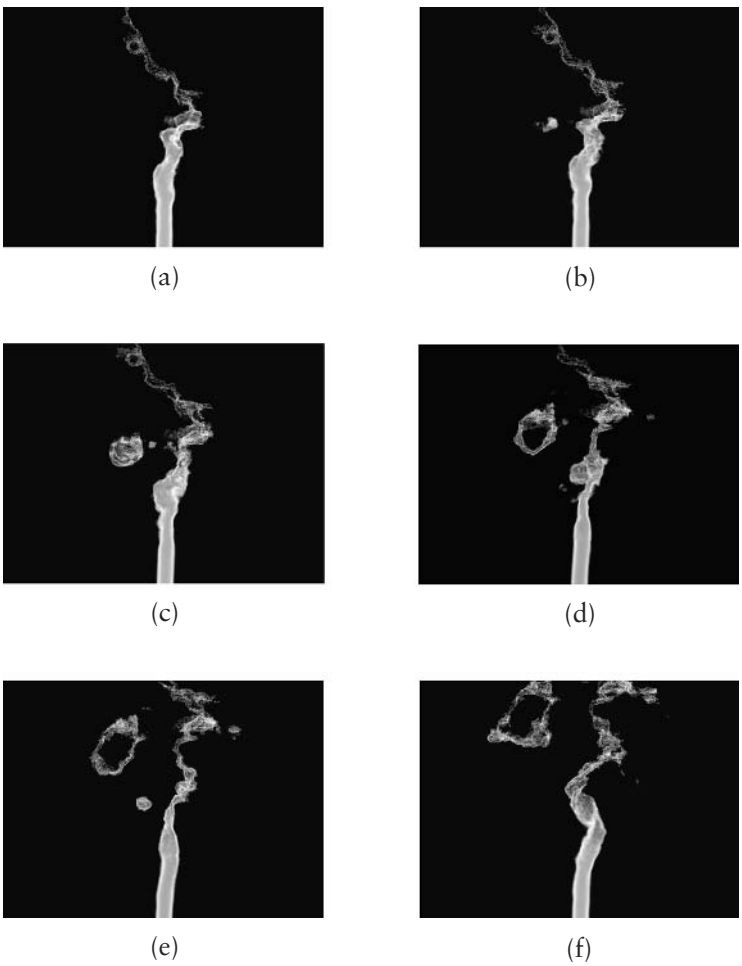
(a) (b)

FIGURE 12.14 Animating smoke column using simple hypertexture.

Team LRN

(a) (b)

FIGURE 12.15 Appearance of "drift" in smoke animation.



(a) (b)

(c) (d)

(e) (f)

FIGURE 12.16 Formation of a smoke ring.

Team LRN

For this reason, we employ two different gain curves. One controls turbulence amplitude near the column as a function of *y*, and the other controls turbulence amplitude far from the column as a function of *y*. The latter curve always lags behind the former, thereby preventing low-lying smoke rings.

Optimization

Since smoke is quite sparse within its sampled volume, it is a good candidate for optimization based on judicious presampling. Tests on the AT&T Pixel Machine with 64 DSP32 processors showed that it took about 8 hours to compute a single frame of a 640 × 480 × 640 volume. This is rather impractical for animation. To speed this up, we precompute the image at a smaller resolution to find out where the smoke lives within the volume. We then do the final computation only within those parts of the volume.

More specifically, we do a preliminary raymarch at one-fourth the final *x*, *y*, *z* resolution. Note that this requires only 1/64 as many density evaluations as would a full computation. At each 4 × 4 pixel, we save the interval along *z* that bounds all nonzero densities. For many pixels, this will be a null interval. Then to be conservative, at each pixel we extend this interval to be its union with the intervals at all neighbor pixels.

We use this image of bounding *z* intervals to restrict the domain of the final raymarching. We have found about a 30-fold speedup using this method—each image now takes about 16 minutes to compute (including the time for the subsampled prepass). Smoke is optimal for this type of speedup for two reasons: (1) since it is sparse, the speedup is great, and (2) since density takes many sample widths to fall off to zero, tiny details are not inadvertently skipped over.

Here is pseudocode for the smoke density function. It's mostly just C code with some unimportant details and declarations left out.

```
smoke_density_function(x, y, z)
{
     /* k1, k2, etc. are the column shape knobs */

     /* rotate z randomly about y, based on noise function */
     /* this creates the impression of random "swirls" */

     t = noise(x,y,z);
     s = sin(t / 180. * PI);
     c = cos(t / 180. * PI);
     z = x * s + z * c;
```

Team LRN

```
    /* once the space is "swirled", create the column of smoke */

    /* 1) phase-shift x and z using turbulence; this varies with time */

    x += k1 * phase_shift(x, y, z);
    z += k2 * phase_shift(x, y, z + k3);

    /* 2) define column by distance from the y-axis */

    rx = (x * x + z * z) * k4;

    /* 3) if inside the column, make smoke */

    if (rx < =1.) {
        rx = bias(rx, k5); /* the basic column shape is */
        s = sin(PI * rx); /* a tube with hollow core */
        return s * s;
    }
    else
        return 0.;
}

phase_shift(x, y, z)
float x, y, z;
{
    /* c1, c2, etc. are the "texture" knobs */

    p[0] = c1 * x + bias(y + .5, .3) * TIME; /* vary with time */
    p[1] = c1 * y + TIME;
    p[2] = c1 * z + c2;
    g = gain(y + .5, c3); /* dropoff with y */

    /* these 3 lines remove smoke rings that are */
    /* too low in y to be physically plausible */

    r = max(0., 1. - (x * x + z * z) * c5);
    gl = gain(bias(y + .5, c4), c3);       /* smoke ring dropoff with y */
    g = g * LERP(gl, r, 1.);

    return g * (turbulence(p, 1., RES) + c6); /* c6 recenters the column */

}
```

TURBULENCE

The turbulence function, which you use to make marble, clouds, explosions, and so on, is just a simple fractal generating loop built on top of the *noise* function. It is not

a real turbulence model at all. The key trick is the use of the `fabs()` function, which makes the function have gradient discontinuity "fault lines" at all scales. This fools the eye into thinking it is seeing the results of turbulent flow. The `turbulence()` function gives the best results when used as a phase shift, as in the familiar marble trick:

```
sin(point + turbulence(point) * point.x);
```

Note the second argument in the following code, `lofreq`, which sets the lowest desired frequency component of the turbulence. The third argument, `hifreq`, is used by the function to ensure that the turbulence effect reaches down to the single pixel level, but no further. I usually set this argument equal to the image resolution.

```
float turbulence(point, lofreq, hifreq)
float point[3], freq, resolution;
{
    float noise3(), freq, t, p[3];

     p[0] = point[0] + 123.456;
     p[1] = point[1];
     p[2] = point[2];

     t = 0;
     for (freq = lofreq ; freq < hifreq ; freq *= 2.) {
         t += fabs(noise3(p)) / freq;

         p[0] *= 2.;
         p[1] *= 2.;
         p[2] *= 2.;
     }
     return t - 0.3; /* readjust so that mean returned value is 0.0 */
}
```

ANTIALIASED RENDERING OF PROCEDURAL TEXTURES

This section describes a way of antialiasing edges that result from conditionals in procedurally defined images, at a cost of one sample per pixel, wherever there are no other sources of high frequencies. The method proceeds by bifurcating the calculation over the image at conditional statements. Where a pixel straddles both true and false branches under its convolution area, both branches are followed and then linearly combined. A neighbor-comparing, antialiased, high-contrast filter is used to define regions of bifurcation. The method proceeds recursively for nested conditionals.

Background

If an image is described by an algorithmic procedure, then in principle there are cases where antialiasing can be done analytically, without supersampling, just by examining the procedure itself.

Generally speaking, there are two sources of high frequencies for procedurally generated images. *Edge events* are caused by conditional statements that do Boolean operations on continuous quantities. For example, there are infinitely high frequencies on the image of a circular disk formed by the following conditional statement when sampled over the unit square:

$$\text{if } (x - .5)^2 + (y - .5)^2 < .25 \text{ then } \textbf{white} \text{ else } \textbf{black}$$

The edges of discontinuity in this image can, in principle, be detected by analyzing the statement itself. Nonedge high-frequency events cannot be detected in this way. For example, to render the image represented by

$$\text{if } \sin(10^5 x) \text{ in}(10^5 y) > 0 \text{ then } \textbf{white} \text{ else } \textbf{black}$$

we need to resort to statistical oversampling or other numerical quadrature methods, since this expression has *inherently* high frequencies.

The Basic Idea

As in Perlin (1985), consider any procedure that takes a position (x, y) in an image as its argument and returns a color. Usually, we create an image from such a procedure by running the procedure at enough samples to create an antialiased image. Where the results produce high variance, we sample more finely and then apply a weighted sum of the results to convolve with a pixel reconstruction kernel.

One common source of this high local variance is the fact that at neighboring samples the procedure follows different paths at conditional statements. No matter how finely we oversample, we are faced with an infinitely thin edge on the image—a step function—between the **true** and **false** regions of this conditional.

Our method is to identify those edges on the image caused by conditional expressions. We do this by comparing neighboring sample values at each conditional expression in the computation. We use the results of this comparison to create an antialiased filter that represents the resulting step function as though it had been properly sampled.

To do this, we can view the procedure as a single-instruction-multiple-data (SIMD) parallel operation over all samples of the image. More precisely, we view the

Team LRN

computation as a reverse Polish stack machine. Items on the stack are entire images. For example, the "+" operator takes two images from the stack, adds them sample by sample, and puts the result back on the stack.

This calculation proceeds in lockstep for all samples. When a conditional is reached, we can evaluate both sides of the conditional and then just disregard the results from one of the branches.

Obviously, this approach is wasteful. Instead, we would like to go down only one path wherever possible. We arrange to do this as follows. We recast each conditional expression using a pseudofunction *ifpos*, so that *expr ifpos* evaluates to 1 when *expr* > 0 and 0 otherwise. Using the *ifpos* operator we can recast any function containing conditionals into an expression

$$expr\ ifpos\ [ToDoIfFalse, ToDoIfTrue]\ LERP$$

where t [a, b] *LERP* is a linear interpolation operation defined as follows: when $t \leq 0$ or $t \geq 1$, *LERP* returns a or b, respectively. When $0 < t < 1$, *LERP* returns $a + t(b - a)$. For example:

$$abs(x) = \text{if } x < 0 \text{ then } -x \text{ else } x$$

can be expressed as

$$x\ ifpos\ [-x\ ,\ x\ ]LERP$$

Over the convolution kernel of any pixel, *ifpos* will return **true** for some samples and **false** for others, creating a step function over the image. We actually return a floating-point value between 0.0 and 1.0 to express the convolution of this step function with each sample's reconstruction kernel. Now the problem of properly sampling edges that have been formed from conditions is completely contained within one pseudofunction.

This is done as follows. When execution of our SIMD stack machine gets to an *ifpos* function, it looks at the image on top of the stack and runs a high-contrast filter h on this image, which examines the neighbors of each sample. If the sample and its neighbors are all positive or all negative, then h returns 1 or 0, respectively. Otherwise, h approximates the gradient near the sample by fitting a plane to the values at neighboring samples. Where there is a large variance from this plane, then supersampling must be done—the procedure must be run at more samples.

But where there is a low variance, then just from the available samples, h can produce a linear approximation to the argument of *ifpos* in the neighborhood of

Team LRN

the sample. It uses this to construct a step function, which it then convolves with the sample's reconstruction kernel. As we shall see, this is a fairly inexpensive procedure.

After the *h* function is run, the samples of the image on the stack will fall into one of three subsets:

$$value \equiv 0$$
$$0 < value < 1$$
$$value \equiv 1$$

We label these subsets *F*, *M*, and *T* (for "false," "midway," and "true"), respectively.

Once *h* is run, we gather up all samples in (*F* union *M*) and run the *ToDoIfFalse* statements on these. Then we gather up all samples in (*T* union *M*) and run the *ToDoIfTrue* statements on these. Finally, we do a linear interpolation between the two branches over the samples in *M* and place the reconstructed image on the stack.

Because each conditional splits the image into subsets, some of a sample's neighbors may be undefined inside a conditional. This means that within nested conditionals, *h* may not be faced with a full complement of neighbors. This happens near samples of the image where several conditional edges intersect. As long as there is at least one horizontal and one vertical neighbor, *h* can re-create the gradient it needs. When there are no longer enough neighbors, *h* flags a "high-variance" condition, and supersampling is then invoked.

More Detailed Description

In this section we describe the flow of control of the algorithm. We maintain a stack of lists of *current_samples*. We start with *current_samples* = ALL samples. At any given moment during execution, samples are examined only if they are in the current list.

(I) When the *ifpos* token is encountered:
Evaluate high-contrast filter for all samples in *current_samples*.
Use the result to make the three sublists: *F*, *M*, *T*.
Push these sublists onto an *FMT* stack.
Push a new *current_samples* list (*M* union *T*).

(II) Continue to evaluate normally, over samples in *current_samples*.
If the *ifpos* code is encountered, recurse to (I).

(III) When the beginning of the *ToDoIfFalse* code is encountered:
Pop the *current_samples* stack.
Push a new *current_samples* list (*F* union *M*).

Evaluate normally until matching *LERP* token is encountered.
If the *ifpos* token is encountered first, recurse to (I).

(IV) When the *LERP* token is encountered:
Pop the *current_samples* stack.
For all samples in *current_samples:*
stacktop[1] := LERP(stacktop[1], stacktop[−1], stacktop[0])
Pop the *FMT* stack.

The High-Contrast Filter

Given a large enough subset of the eight neighbors around a sample, we can use the value at the sample and its neighbors to construct an approximation to the integral under the sample's reconstruction kernel in the area where *ifpos* evaluates to 1.

Let $f(x, y)$ denote the value of the expression at any small offset (x, y) from the sample. First we do a least-squares fit of the differences in value between the sample and its neighbors to approximate the x and y partials. If the variance is high for either of these two computations, then we give up and resort to supersampling the procedure in the neighborhood of the sample.

Otherwise, we can use the linear approximation function $ax + by + c = 0$ near the sample, where $c = f(0,0)$, and a and b approximate the x and y partials of f, respectively.

We want to find out the integral under the sample's reconstruction kernel of the step function

$$\text{if } f(x, y) > 0 \text{ then 1 else 0}$$

We assume a circularly symmetric reconstruction kernel.

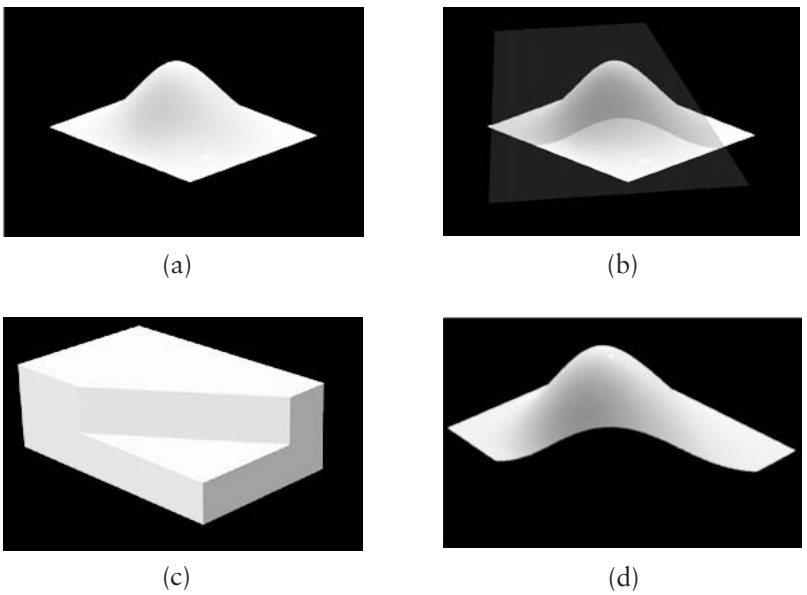
The problem at this stage is illustrated in Figure 12.17. Figure 12.17(a) shows the sample reconstruction kernel, Figure 12.17(b) shows the intersection of this kernel with a linear approximation of the function near the sample, Figure 12.17(c) shows the step function induced by this linear function, and Figure 12.17(d) shows the region under the kernel where this step function is positive.

Since we assume the kernel is circularly symmetric, finding the convolution reduces to a one-dimensional problem. We need only compute the perpendicular distance from the sample to the step and then use a spline fit to approximate the integral under the kernel in the positive region of the step function.

To do this, we compute the magnitude of the linear gradient

$$|f'| = \sqrt{(a^2 + b^2)}$$

Team LRN



(a) (b)

(c) (d)

FIGURE 12.17 Reconstruction kernel.

Then we calculate the distance from the sample to the line of the image where this function equals zero by

$$d = f(x,y) / |f'|$$

Finally, we approximate the integral of the reconstruction kernel by a cubic spline:

$$\text{if } t < -.5 \text{ then } 0 \text{ else if } t > .5 \text{ then } 1 \text{ else } 3t + .5)^2 - 2(t + .5)^3$$

Examples

Figure 12.18(a) through 12.18(f) shows a succession of steps in the creation of a simple procedural image that requires two levels of nested conditionals. Each image is on the top of the stack during a particular snapshot of execution.

The image is formed by doing a conditional evaluation of the circular disk (Figure 12.18(a)):

$$(x - .5)^2 + (y - .5)^2$$

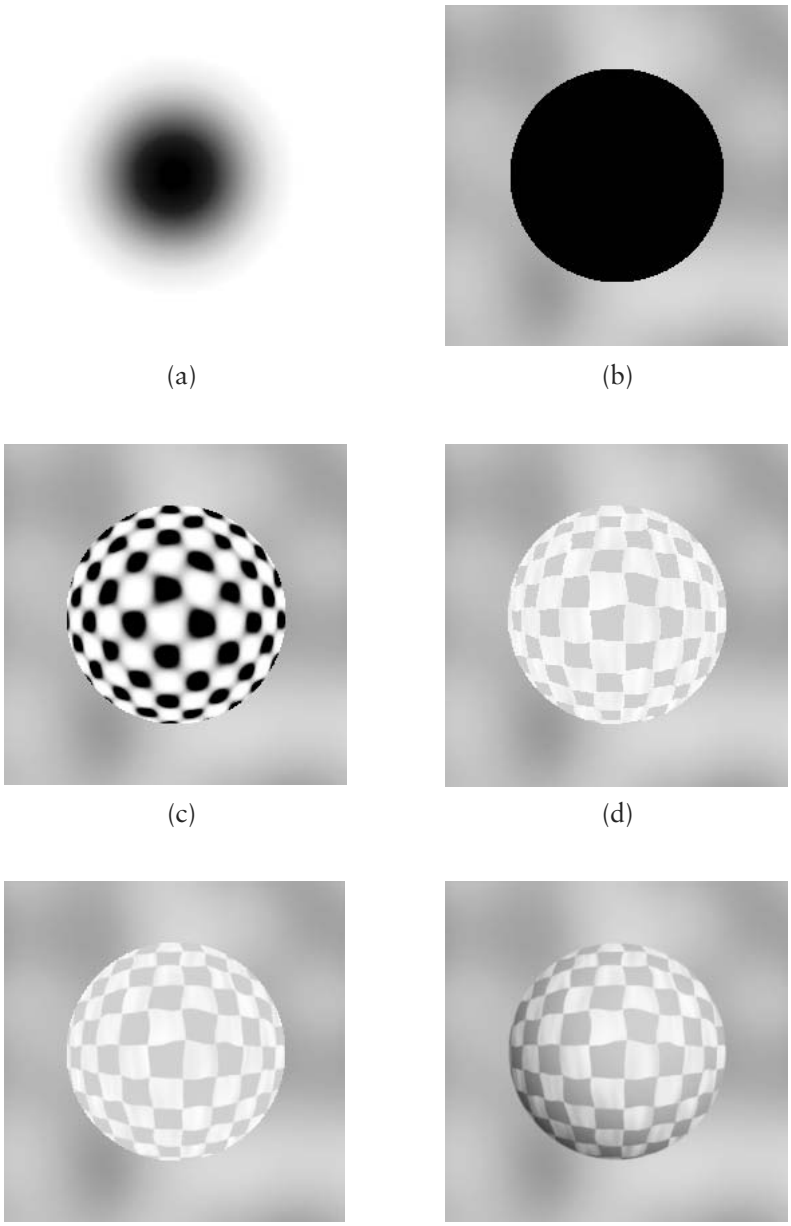


FIGURE 12.18 Creation of a simple procedural image that requires two levels of nested conditionals.

Then $.3^2$ is subtracted, a conditional is done, execution bifurcates, and *noise* (Perlin 1985) is applied to the outside of the disk (Figure 12.18(b)). Then $\sin(10x + noise(x, y)/10)\sin(10y + noise(x, y)/10)$ is evaluated inside the disk (Figure 12.18(c)), and another conditional is done. A constant is applied to the negative areas, and $.5 + noise(3x, y)$ is applied to the positive areas (Figure 12.18(d)). Finally, Figure 12.18(e) and 12.18(f) show the successive reconstruction of the two nested levels of conditionals.

To Sum Up

The previous method recasts image synthesis as a set of recursively bifurcating SIMD processes over image samples. In this way we were able to do true procedural antialiasing of edges.

This approach differs from previous approaches in that it finds and resolves sampling problems by using information contained in the image synthesis algorithm itself, usually without resorting to supersampling. Whether such methods can be applied to the general image synthesis problem is an open question.

SURFLETS

Instead of regenerating hypertexture procedurally all the time, it is sometimes useful to "cache" it in some format for display. For this purpose we want rapid and accurate display, as well as a good ability to do light propagation—including self-shadowing and diffuse penumbra.

This section describes a useful intermediate representation for procedurally generated volumes. The approach is based on a sparse wavelet representation and is particularly suitable for the sorts of free-form surfaces generated by hypertextural algorithms.

Other sorts of complex volumetric data can also be conveniently represented using this approach, including medical data (CT and MR) and meteorological, geological, or molecular surface data.

A *surflet* is a flavor of wavelet that can be used to construct free-form surfaces. Surfaces obtained by evaluating scalar volumes or procedural volumetric models such as hypertexture can be stored in the intermediate form of surflets. This allows us when rendering such surfaces to (1) do self-shadowing, including penumbras, from multiple light sources, and (2) have the option at rendering time to either (a) further refine the surface locally from the volume data or (b) do a bump texture approximation or any mixture of (a) and (b). Thus issues of lighting placement and level-of-details differences can be resolved at a postprocessing stage, after the bulk of

Team LRN

the computation is completed. This is work I did mainly with Benjamin Zhu and is based in part on earlier work that I did together with Xue Dong Yang.

First we will define the surflet model and show a particular implementation. Then we will discuss how to find visible surfaces. We will show how, by using surflets, we can decide locally at rendering time whether true volumetric surface refinement or shading approximation should be performed. Finally, we will introduce a multiresolution, hierarchical modeling of surflets and describe a way to do self-shadowing with penumbra.

Introduction to Surflets

Volume visualization studies the manipulation and display of volume data. Depending on the intermediate representation between raw data and final display, volume visualization techniques can be classified as surface-based techniques (or surface modeling) (Lorensen and Cline 1987, 1990; Wyvill, McPheeters, and Wyvill 1986) and volume rendering (Drebin, Carpenter, and Hanrahan 1988; Hanrahan 1990; Kajiya and Von Herzen 1984; Levoy 1988, 1990a, 1990b, 1990c; Sabella 1988; Westover 1990). Methods in the first category approximate surfaces of interest (determined by thresholds) by geometric primitives and then display these primitives, whereas volume rendering directly operates on the volume data and renders the scenes.

Surface modeling is the main topic of this discussion. Since surfaces offer a better understanding of the external structure of volumes, accurately reconstructing surfaces from volumes and subsequently rendering the surfaces efficiently is very important in practice. Surface modeling techniques differ from one another in that different primitives can be used to approximate the surfaces of interest. Keppel (1975) and Fuchs (Fuchs, Kedem, and Uselton 1977) construct contours on each 2D slice and connect contours on subsequent slices with triangles. The cuberille model uses parallel planes to create surfaces (Chen et al. 1985; Gordon and Reynolds 1985; Hoehne et al. 1990). The marching-cube algorithm uses voxel-sized triangles to approximate surfaces (Cline, Lorensen, and Eudke 1988; Lorensen and Cline 1987, 1990; Wyvill, McPheeters, and Wyvill 1986). Bicubic patches can also be used to reconstruct surfaces (Gallagher and Nagtegaal 1989).

All of these techniques share a common characteristic: the geometric primitives involved in surface reconstruction are in an explicit form. This leads to an interesting question: Can we define the surfaces of interest in an implicit form? If so, what are the advantages of using the implicit representation?

In the following sections, a "surflet" model is introduced to define free-form surfaces. Those sections will describe methods to find visible surfaces, perform selective

Team LRN

surface refinement, and do self-shadowing with penumbra. A multiresolution, hierarchical modeling of isosurfaces is sketched. Experimental results are shown to demonstrate the benefits of adopting this new model.

Surflets as Wavelets

As discussed earlier in the section on *noise*, a wavelet (Mallat 1989a, 1989b) is a function that is finite in extent and integrates to zero. Decomposing signals or images into wavelets and then doing analysis on the wavelet decomposition is a powerful tool in signal and image recognition.

We use a specific kind of continuous wavelet, which we call surflets, to approximate surfaces. We use a summation of surflets that together define a function over R^3 so that the desired surface is closely approximated by the locus of points where (i) the summation function is zero but where (ii) the function's gradient magnitude is nonzero. The general intuition here is that each surflet is used as a free-form spline that approximates a little piece of localized surface. All of these surflets sum together to form isosurfaces of interest.

We define each surflet as follows: let $\vec{p} = [x, y, z]$ be a sampled location in space and let r be its sampling radius. Then if the sample at $\vec{p}$ has a value of d and a gradient vector of $\vec{n}$, we define a wavelet approximation to the sample at $\vec{x} = [x, y, z]$ near $\vec{p}$ by

$$\prod drop\left(\frac{x_i - p_i}{r}\right) \times \sum n_i(x_i - p_i)$$

where

- $\vec{x}$ is any point in R^3
- $\vec{p}$ is the wavelet center
- $\vec{n}$ is the wavelet gradient
- r is the wavelet radius
- i varies over the three coordinates
- $drop(t)$ is defined as for the *noise* function: $2t^3 - 3t^2$

At points with distance greater than r from $\vec{p}$ we assign a value of zero.

$[\vec{p}, d, \vec{n}]$ defines a surflet. Since each sample at $\vec{p}$ cannot affect any sampled location farther than r away from its center, the wavelet contribution is localized.

Team LRN

It should be pointed out that other continuous functions might also work well, as long as each surflet has a limited extent in terms of its wavelet contribution, and this contribution drops from 1.0 down to 0.0 monotonically. The cubic drop-off function is chosen due to its simplicity.

In practice, we define the surflets on a rectangular grid. When surflets are refined (see ahead), the size of the sampling grid is successively halved. We generate surflets from a functional sampling with the following steps:

- Subtract the desired isovalue from all samples.
- Find samples that have any six-connected neighbor with density of opposite sign.
- Approximate surflet normal $\vec{n}$ by density difference between neighbors.
- Approximate surflet normal $\vec{n}$.
- Use $\vec{n}$ to locate center $\vec{p}$ near the sample.

We use the normal to locate the surflet center as follows:

- Approximate surflet normal $\vec{n}$ by taking density difference between neighbor samples along each coordinate axis.
- Use $\vec{n}$ to locate surflet center $\vec{p}$:

$$\vec{p} = \vec{x}_{\text{sample}} - d_{\text{sample}}\vec{n}$$

Note that we end up with something very closely related to the *noise* function (Perlin 1985). In fact, the *noise* function can be viewed as the surflet decomposition of a random density function sampled at a rectangular grid.

Finding Visible Surfaces

Because surflets are well-defined analytic functions, it is possible to find the intersection of a ray with the zero surface of a summation-of-surflets function.

Since the contribution from each surflet has a limited extent, for each point $\vec{x}$ in R^3 only a finite number of surflets will have a nonzero wavelet contribution to $\vec{x}$. Two alternatives are possible for finding a ray-surface intersection. A numerical method such as binary division or Newton iteration can guarantee convergence to the true intersection, given initial guesses close to the actual solution. However, since all these numerical methods involve iterative root finding, they can be quite slow. We instead use a faster approximation method.

Team LRN

As in cone tracing (Amanatides 1984), we trace a fat ray (a cone) into the scene. We represent each surflet by a sphere whose radius is the surflet's r times a constant factor and whose center is $\vec{p}$. We shrink the cone ray into a thin line and grow all the spheres by the cross-sectional width of the cone at the surflet. This is equivalent to the computation in Amanatides (1984). Since perspective projection is used, the spheres that approximate the surflets become variously sized. Each sphere is given a weight according to how far along its radius it intersects the ray as well as the magnitude of its gradient. Spheres that mutually intersect each other within the ray form a local bit of surface. We use a scheme similar to the fragment merging in Carpenter's A-buffer algorithm (Carpenter 1984) and use the weighted average of the surflet normals as the surface fragment normal. All such surface fragments are evaluated from front to back along the ray as opacity accumulates.

Shading is done as in Duff (1985). We stop raymarching when the opacity reaches totality. Because we render one surflet at a time (as opposed to one ray at a time), our surface approximation method is view dependent, in contrast to the numerical approach, which is view independent.

Visible surfaces can be rendered efficiently by using a depth-sorting algorithm similar to the Z-buffer algorithm. More efficiency can be derived by using the selective surface refinement described in the following section and the hierarchical surflet modeling described in the section "Constructing a Surflet Hierarchy."

Selective Surface Refinement

A sampled location is defined as a singularity if, along any coordinate axis, it is inside the surface while its two neighbors are outside the surface, or it is outside the surface while its two neighbors are inside the surface.

We generally need to refine a surflet only when any one of the following conditions is true:

- Due to perspective, the surflet is large with respect to a pixel on the viewing plane.
- The surflet normal is nearly perpendicular to the line of sight, thus presenting a silhouette edge.
- The surflet is a singularity.

However, if only the first condition is satisfied, then a surflet, no matter how large, can be visually approximated by normal perturbation alone.

Team LRN

By using selective surface refinement, we can greatly reduce the rendering time. Since surface refinement adapts to the complexity of a scene, different parts of the scene can be rendered at different resolutions. This is very attractive in practice. We have observed that the number of surflets involved in rendering decreases by more than 50% when we start at a low resolution and perform selective surface refinement at the next higher resolution than when we start directly at the next resolution level.

A Surflet Generator

Surflets can be generated from either sampled scalar data or procedural volumetric functions such as hypertexture. In the first case, we read in digitized samples from a regular grid; in the latter case, we evaluate samples on a regular grid. One orthogonal 2D slice is processed at a time. We always keep three slices, which helps us detect surflets as well as singularities.

After each slice has been processed, we find those samples that have any neighbors with density value on the opposite side of the isosurface density. At a candidate sample located at point $\vec{p}$, we approximate the normal gradient $\vec{n}$ by the density difference d between the samples and its neighboring samples along each coordinate axis. Forward difference, central difference, or a mixture of both (Hoehne et al. 1990) can be applied to get gradients. If the distance from the sample to the surface is less than 0.5, then $[\vec{p}, d, \vec{n}]$ defines a surflet. This 0.5 bias is determined empirically to compromise between two constraints: (1) If the distance is too small, we do not have enough surflets to interlock with each other, and consequently we might get undersampling. (2) If the distance is too big, then we will have too many surflets. This will create a huge intermediate structure to store surflets, and the rendering time will be too high.

For those samples that are singularities, each one is treated as eight different samples in eight octants; their gradient vectors and distances to the isosurface are evaluated. Therefore, a maximum number of eight surflets can be generated. Each singular surflet will have an appropriate flag tagged in its data structure, indicating which octant it contributes to.

Constructing a Surflet Hierarchy

Up to now we have described surflets limited to a single resolution. However, it would be more desirable to create surflets iteratively from lower resolutions to higher resolutions. A hierarchical surflet model not only can provide the freedom to

Team LRN

render surfaces at arbitrary levels of detail but also can avoid unnecessary details that would be washed out due to shadowing and so on.

We construct a surflet hierarchy as follows: Since only the isosurfaces are interesting to us, a candidate set of potential surflets is created at each resolution as in the previous section. Each candidate set is generally very small as compared to the number of samples in the volume. All surflets at the lowest resolution level are detected and collected. We compute the image at this resolution based on the wavelet approximation formula. Then we proceed to the next higher level and subtract the image at this level from the image at the previous level. Because we have precomputed the surflet candidate set at this level, the computation only involves those potential surflets. Those samples less than $r/2$ away from the isosurface are classified as surflets, where r is the sampling radius at this resolution. We repeat the same procedure and subtract images at lower resolutions from higher resolutions. When the sampling resolution equals the size of the sampling grid, a surflet hierarchy is created.

Self-Shadowing with Penumbra

Let us assume for simplicity that there is a single light source in the scene. We can then handle multiple light sources as the summation of the shading contribution from each light source.

We want to render surfaces with self-shadowing and penumbra. It is theoretically impossible to use a point source to create true penumbra, although some filtering techniques might create quite vivid penumbra effects (Reeves, Salesin, and Cook 1987). We instead use a light source of a certain shape, such as a spherical light source (or a rectangular light source). The portion of light source visible from any surflet is computed and used to determine the brightness at a surflet. An approximation of this portion is sketched here.

Each surflet in the scene is approximated by a sphere whose radius r is the sampling radius and whose center is given by $\vec{p}$. For each visible surflet, we trace a fat shadow ray from the surflet center to the light source. The spread angle of the fat ray is approximated by R/D, where R is the radius of the spherical light source and D is the distance from the surflet center to the center of the light source. We perform cone tracing to determine shadowing and penumbra in the same way that we found visible surfaces. We shrink the cone ray into a thin line and grow the spheres accordingly. The portion of the shadow ray that any surflet blocks is determined by how far along its radius the surflet intersects the ray. To make the approximation more accurate, spheres that mutually intersect each other within the ray are split into small,

Team LRN

disjoint fragments. Portions of the shadow ray blocked by all these small segments are computed and summed to find the proportion of light blocked by this piece of surface.

We compose the portion of the shadow ray blocked by isosurfaces by a method similar to the image composition scheme in Duff (1985). The shadow ray marches through the scene until (1) the blocked portion reaches totality or (2) the shadow ray reaches the light source. In the first case, the surflet that initiates the shadow ray is in full umbra. In the latter case, we subtract this blocked portion from 1.0 to get the surflet illumination. Clearly, the surflet is in penumbra if its illumination level is less than 1.0, and the surflet is not in shadow if its illumination equals 1.0.

To speed up the rendering, we can render surflets in shadow at lower resolutions. For example, we can do shadow detection at low resolutions and do selective surface refinement only for those visible surflets not in shadow. A major advantage of this scheme is that important surface details are not left out, while unnecessary details hidden by the shadow are not given much attention. This can produce softer shadows, as well as increase rendering speed.

Discussion

The surflet model has a number of advantages over other surface-based techniques. First, it has an implicit form. Although we approximate surflets with spheres in our implementation, it does not mean that this is the only choice. On the contrary, it is not clear to us whether this is the best way. For example, with special hardware, numerical root finding can be more accurate and more promising. We have also started to experiment with ellipsoid-like primitives to approximate surflets.

Second, the surflet model provides a convenient way to do hierarchical modeling of surfaces and selective surface refinement due to its implicit form. This feature cannot be found in many existing surface modeling methods. Adaptive sampling gives us the power to avoid unnecessary details while preserving important surface subtleties.

Third, the representation has a compact structure. Our experiments indicated that using surflets takes only 25% to 50% of the storage of marching cubes at the same resolution.

Fourth, isosurfaces can be rendered in parallel with surflets. Since for any point on the isosurfaces there are only a limited number of surflets determining the zero crossing, surflets are amenable to either a parallel implementation or an implementation with distributed computing.

Team LRN

Fifth, surflets can be integrated with wavelets in a straightforward manner to yield a combined model of surface modeling and volume rendering. If we delay thresholding until the rendering stage, we can generate wavelets by subtracting the low-resolution signals from the high-resolution signals with the same kind of hierarchical modeling as in the section "Constructing a Surflet Hierarchy." Depending on our need, we can do either volume rendering or thresholding followed by surface rendering at rendering time. This integrated approach is attractive, since it reduces the difference between rendering surfaces and rendering volumes. However, it differs from conventional volume rendering (Drebin, Carpenter, and Hanrahan 1988; Levoy 1988, 1990c) in that an intermediate data structure, as well as hierarchical modeling, is introduced to speed up the process. Thresholding can still be used at rendering time to distinguish the rendering from volume rendering. Moreover, it is possible to have volume details and surface details in the same scene.

Conclusion

Surflets are a free-form modeling of isosurfaces. This model is attractive in that it provides a convenient way to do shadowing, selective surface refinement, and hierarchical modeling. Moreover, it requires much less storage than other volume-to-surface methods and allows considerable freedom for a particular implementation. Finally, it encourages an integrated approach for surface modeling and volume rendering.

This surflet model is still quite empirical. Although it is quite intuitive and supported by image synthesis theory (Grossman and Morlet 1984; Mallat 1989a, 1989b), in many places we have had to tune the parameters to make the rendered images more realistic.

There is a lot of promise in integrating surface modeling with volume rendering. This kind of hybrid is very promising for hierarchical modeling of surfaces, since hierarchical modeling of volumes is more general than that of surfaces.

FLOW NOISE

In recent work with Fabrice Neyret (Perlin and Neyret 2001) we have been modifying the *noise* function so that it can give a suggestion of swirling and flowing time-varying textures.

Flow textures that use noise can look great, but they often don't "flow" right, because they lack the swirling and advection of real flow. We extend *noise* so that

Team LRN

shaders that use it can be animated over time to produce flow textures with a "swirling" quality. We also show how to visually approximate advected flow within shaders.

Rotating Gradients

Remember that we can think of the *noise* function as a sum of overlapping pseudo-random wavelets. Each wavelet, centered at a different integer lattice point (i, j, k), consists of a product of a weight kernel K and a linear function $(a, b, c)_{i,j,k}$. K smoothly drops off away from (i, j, k), reaching 0 in both value and gradient at unit distance. Each $(a, b, c)_{i,j,k} = a(x - i) + b(y - j) + c(z - j)$, which has a value of 0 at (i, j, k). The result of summing all these overlapping wavelets, *noise* has a characteristic random yet smooth appearance.

Our modification is to rotate all the linear vectors $(a, b, c)_{i,j,k}$ over time, which causes each wavelet to rotate in place. This process is illustrated in Figure 12.19.

Because all the (a, b, c) vectors were uncorrelated before the rotation, they will remain uncorrelated after the rotation, so at every moment the result will look like noise. Figure 12.20 illustrates this difference.

Yet over time, the result will impart a "swirling" quality to flow. When multiple scales of noise are summed together, we make the rotation proportional to spatial frequency (finer noise is rotated faster), which visually models real flow.

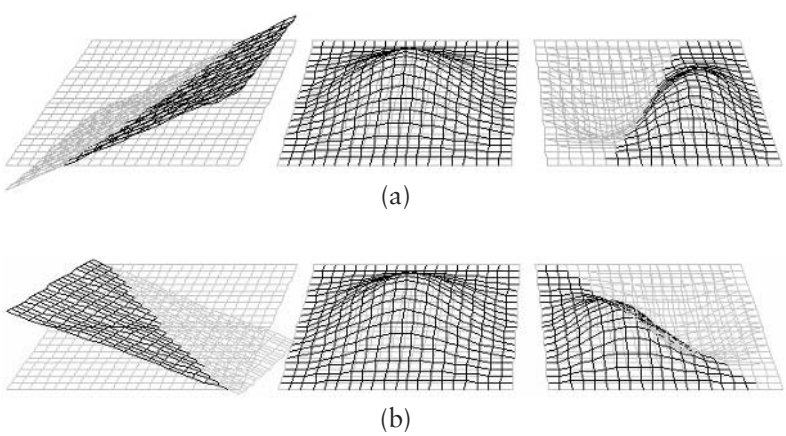


FIGURE 12.19 (a) Gradient * kernel ⇒ wavelet; (b) rotated gradiant * kernel ⇒ rotated wavelet.

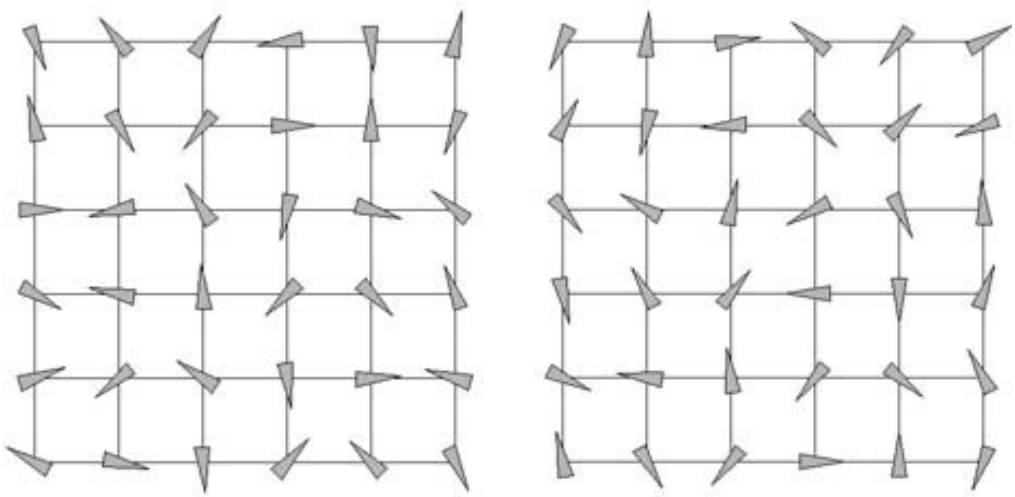


FIGURE 12.20 (a) Before rotation; (b) after rotation.

Pseudoadvection

Beyond swirling, fluids also contain advection of small features by larger ones, such as ripples on waves. This effect tends to stretch persistent features (e.g., foam), but not newly created ones (e.g., billowing), to varying degrees, according to their rate of regeneration, or *structure memory M*.

Traditional Perlin turbulence is an independent sum of scaled noise, where the scaled noise is

$$b_i(x) = noise(2^i x)/2^i$$

and the turbulence is

$$t_N(x) = \sum_{i=0}^{N} b_i(x)$$

This can define a displacement texture

$$color(x) = C(x + It_N(x))$$

where C is a color table and I controls amplitude. Our pseudoadvection displaces features at scale $i + 1$ and location x_0 in the noise domain to $x_1 = x_0 + k\ t_i(x_0)$, where k is the amplitude of the displacement (see below). For small displacements, this can be approximated by $x_1 - k\ t_i(x_1)$, so displacement k is proportional to an amplitude I specified by the user. We can scale this displacement by a "structure memory" factor M. We can simulate passive structures, which are totally advected, when $M = 1$, or very active structures, which are instantaneously generated and therefore unstretched, when $M = 0$. Our advected turbulence function is defined by modifying the scaled noise to

$$b_i(x) = b(2^i\ (x - IM\ t_{i-1}(x)))/2^i$$

and using this to construct the sum

$$t_N(x) = \sum_{i=0}^{N} b_i(x)$$

Results

Many flow textures can be created. A few are shown in Figures 12.21–12.24.

PROCEDURAL SHAPE SYNTHESIS

Recently, I've been working with Luiz Velho and others on multiresolution shape blending (Velho et al. 2001). In this approach, you conceptualize surface shapes as textures and then use multiresolution techniques to smoothly blend those shapes together, just as you might blend one texture into another. For example, just as Peter Burt (1983) originally showed that you can smoothly blend two images together by separately blending each of their multiscale components, you can do similar things with textural surface deformations. Two very interesting images we created using these techniques are shown in Figures 12.25 and 12.26.

TEXTURAL LIMB ANIMATION

In this section we borrow notions from procedural texture synthesis and use them to control the affect of humanlike figures. Stochastic *noise* is applied to create time-varying parameters with well-controlled statistics. We then linearly blend these parameters and feed the results into a motion system. Potential uses of this technique include role-playing games, simulated conferences, "clip animation," and simulated dance.

Team LRN

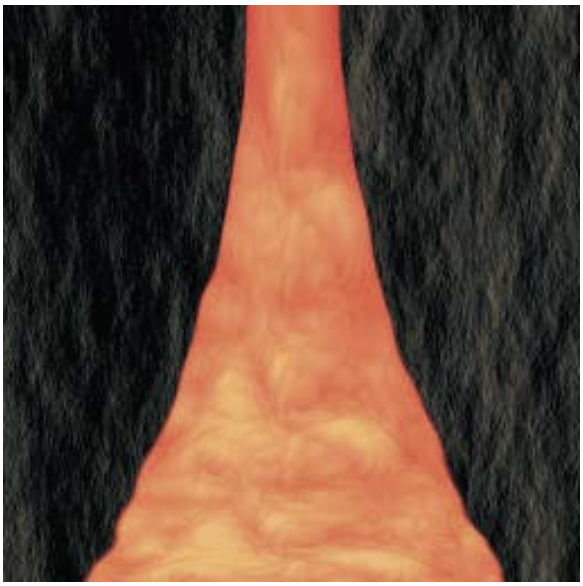


FIGURE 12.21 Lava flow.

FIGURE 12.22 Waterfall.

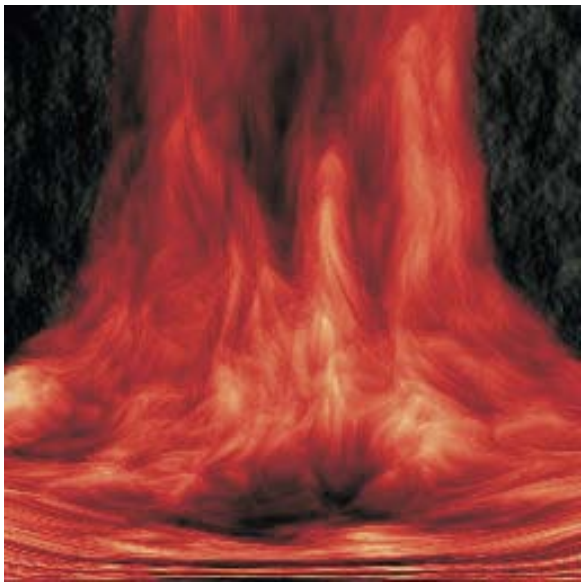


FIGURE 12.23 Waterfall with lava flow texture.

FIGURE 12.24 Swirling clouds using flow noise.

Team LRN

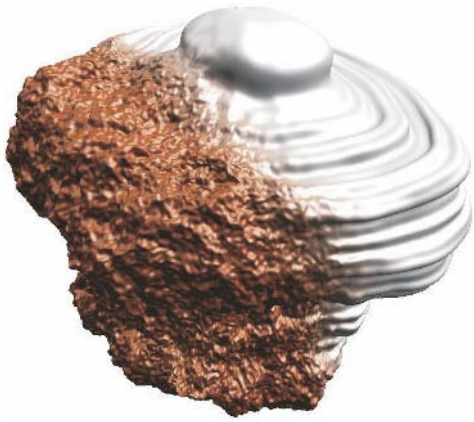


FIGURE 12.25 Rock blending into a screw shape.

FIGURE 12.26 Corroded skull blending into a fresh one.

Team LRN

Introduction to Textural Limb Motion

In simulated environments we often want synthetic cooperating characters to respond to each other and to a changing environment in real time. Because simulated worlds are often used primarily as a means of communication (in contrast to, say, robotics simulation), we are often not so concerned with specific task completion as with conveying emotional messages through motion or gesture.

This situation comes up in role-playing games, where game players want their "Avatars" (Stephenson 1992) to show a particular affective response to a person or event in the immediate environment.

Similarly, emotive gesturing would be useful for monitoring a shared electronic "talk" conference. You could model the participants as physical gesturing characters around a conference table. At a glance you could tell who is talking to whom, who is entering and leaving the discussion, and who is paying attention to whom.

Also, it is useful to have "clip animation," much like clip art, where, for example, an entire crowd of people reacts in a particular way to a character or event. We are not so concerned with their particular motions, but with the sense that "the crowd went wild" or "they respectfully parted to let her pass."

Dance is another instance where we're primarily interested in the *affect* of gesture. Music conveys emotional meaning, not literal meaning. It would be useful to be able to generate simulated dancing characters to music, by working on the level of "Now I want the dancer to convey this combination of emotions and reactions to the other dancers."

The system described here aims at allowing the building of and use of gesture on the level of affective communication. It sidesteps many difficult issues faced by systems that address robotic problems such as "Pick up the glass of water and drink it." The work described here can be plugged into such systems as a modifying filter, but is independent of them.

Road Map

The structure of this section is as follows. First will come a description of some prior and related work by others. This will be followed by an outline of the basic approach, illustrated with a simple example.

After this we will describe textural gesture and the structure of the system—what the different layers and modules are and how they communicate. We will follow this with some examples of the system in use. This section will conclude with a discussion of future and ongoing work.

Team LRN

Related Work

A number of researchers have done work complementary to that described in this section. Badler has specified movements in a goal-driven way and then fed those goals to a simulator (Badler, O'Rourke, and Kaufman 1980). Calvert sampled human figure motion and used Labanotation (a form of dance notation) to create an animated deterministic stick figure (Calvert, Chapman, and Patla 1980). Miller (1988b) has applied synthesis techniques to worms.

Waters (1987) did procedural modeling of faces. Deterministic methods for dance have been described by Yang (1988). Actor/script-based systems for crowds (flocks and herds) were first described by Reynolds (1987). Goal-based movement was described by Badler, O'Rourke, and Kaufman (1980).

Physically based systems for jointed motion are very powerful, although they can be difficult to control and specify (Girard and Maciejewski 1985). Fusco and Tice (1993) take a sampling approach, digitizing sequences of actual human movement and using those to drive animated figures.

Basic Notions

The basic idea is that often we don't care what a gesture is actually doing, so long as it is conveying particular emotional information. For example, when someone is explaining something, his or her arms will wave about in a particular way. Different people, and the same people in different emotional states, will do this in a way that can be defined statistically. Those watching do not care exactly where the hands or arms of the speaker are at any given moment. Instead, they watch the rhythm, the range of motion, the average posture, and the degree of regularity or irregularity of movement.

Our approach is to create a nondeterministic "texture" as a synthesis of scalar control parameters and to use these parameters to drive a motion description subsystem.

Stochastic Control of Gesture

The key innovation is the use of stochastic controls to specify gestural affect. But this needs to be placed in an appropriate system to be useful. This section consists of two parts. First we will describe stochastically defined gestures and then the system in which this technique is embedded.

Any animatable character has some N input parameters that control its motion. At any moment, the character can be thought of as residing at a point in an

Team LRN

N-dimensional unit cube. Each dimension spans the lowest to the highest value of one parameter.

We define a *textural gesture* as a stochastically defined path of motion within this cube that has constant statistical properties. Many gestures of a human figure can be defined by assigning to each joint angle a value for the triple `[center,amplitude,frequency]` and using the following procedure at the `time` of each frame:

```
center + amplitude * noise (frequency * time)
```

This procedure was used for most of the gestures in this section. It allows control over average position and frequency of undulation, while conveying a "natural" quality to all motion. This procedure has a constant statistical quality over time and therefore can be thought of as a fixed point in a gesture space induced over the *N*-cube space.

Several specific and interesting relationships came out of playing with these parameters. For example, I found that natural arm gestures generally resulted when elbow joints were moved with twice the frequency of shoulder joints. Varying too far from this ratio produced unnatural and artificial-looking gestures. This might be related to the fact that the weight of the entire arm is approximately twice the weight of the lower arm alone. The output of these procedures is combined via linear blending and fed into a kinematic model. In this way the single gestural "points" of the induced gesture space are used to traverse convex regions in this space.

The system provides the user with a general-purpose interpreted language for defining new gesture textures, in the spirit of Stephenson (1992). Surprisingly, almost all gestures built using the system could be defined by linearly transformed *noise* of the various joint angles. For example, the bottom row of Figure 12.27 shows a transition from a "fencing" gesture to a "conducting" gesture. This illustrates sliding along a line within the induced gesture space.

The System

The texturally defined parameters feed into "scene description modules" (SDMs). An SDM can represent an animated character, a group of animated characters, or some physical object(s) in the scene. Each SDM knows about kinematics, static constraints, dynamics, and so on (whereas the texture module does not). This separation allows a very simple, high-level control of emotive qualities and makes those qualities very easy to modify. An SDM can take as input a set of scalar parameters and generally outputs scene transformations (matrices) and geometry to be rendered. Also, one SDM can be dependent on the matrices computed by another. For clarity

Team LRN

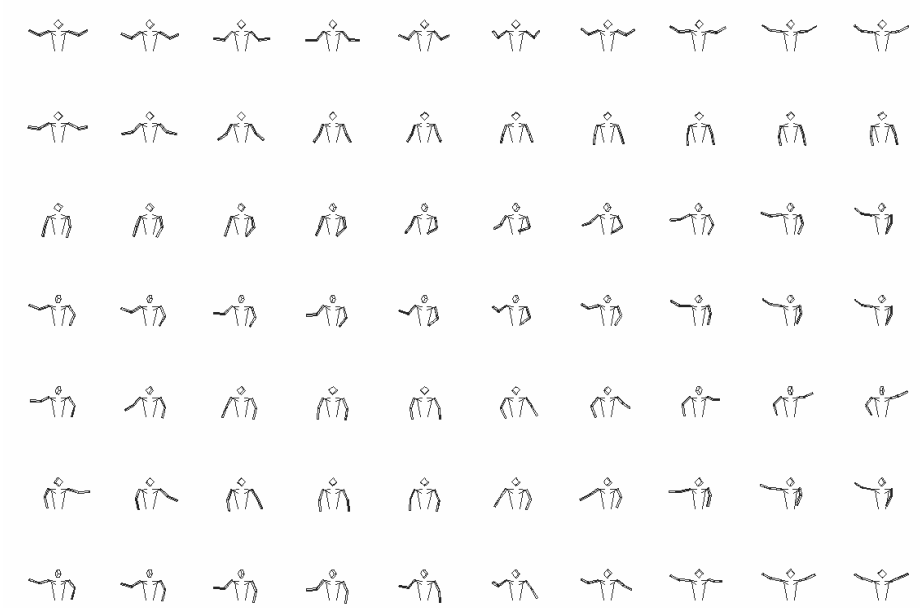


FIGURE 12.27 Gesturing man.

in this section, we will define a *parameter* as a scalar value that controls the movement of an SDM and a *gesture* as a procedure that outputs varying values over time of a set of scalar parameters that feed into an SDM.

The system evaluates a frame of animation in several layers. At the top is a goal determination module (GDM). The GDM is responsible for determining, at each frame of animation, a weight for each gesture.

Below this we have the stochastic procedures for creating individual gestures. A gesture *g* is invoked if its weight is nonzero. Each gesture creates values over time for some set of parameters, as described in the previous section.

Once all gestures have been evaluated, we perform for each parameter a weighted sum of its values within each gesture for which it is defined. All parameter values are then normalized by their respective accumulated weights.

Finally, all SDMs are run for this frame. The transformation matrices produced by the various SDMs are available to an analyzer, which uses them to produce scalar variables that can be fed back into the GDM. For example, as character B physically

Team LRN

approaches character A, the analyzer can evaluate their mutual distance and feed this back to the GDM to influence character A to increase the weight of a gesture that conveys a sense of, say, "being sad" or "waving to character B."

EXAMPLES

Figure 12.27 shows a gesticulating man at various points in his gesture space. The top row shows him at a fixed "conducting" gesture point. If held at this point, he will continue to convey exactly the same affect forever, while never actually repeating the same movement twice. The second row shows a transition between this point and a "sad" gesture point—downcast with relatively slow movement, sloping shoulders, and low energy.

The third row shows a linear transition to a "fencing" gesture point. This point is held throughout the fourth row. In the fifth and sixth rows the man switches to fencing the other way and then back again. Finally, in the bottom row he transitions back to the original gesture point.

One important thing to note is that every point in the transition between gestures produces reasonable motion. This is logical, since the statistical properties of motion during these transitions are still under tight control.

A reasonable use to this approach would be to do statistical analysis/resynthesis of human motion. This would involve analyzing statistics from real human figure motion and turning these into sum-of-noise descriptions. These gesture "points" would then be added to the system.

TEXTURE FOR FACIAL MOVEMENT

In this section we apply texture principles to the interactive animation of facial expression. Here the problems being addressed are "How do we make an embodied autonomous agent react with appropriate facial expression, without resorting to repetitive prebuilt animations?" and "How do we mix and transition between facial expressions to visually represent shifting moods and attitudes?"

By building up facial expression from component movements, and by approaching combinations of these movements as a texturing problem, it is possible to use controlled *noise* to create convincing impressions of responsiveness and mood.

This section is structured as follows. After giving a background and discussing some related work, we will introduce a simple layered movement model. Then we will describe the mechanism that allows animators to build successive abstractions.

Team LRN

This is followed by examples of the use of controlled *noise* to build up elements of autonomous facial movement. The section concludes with a discussion of some future work.

Background

Much human communication is conveyed by facial expression (Faigin 1990). One of the limitations of computer-human interfaces is their inability to convey the subtleties we take for granted in face-to-face communication. This concept has been well described in speculative fiction on the subject (Stephenson 1992). Toward this end, Parke (1982) and others have made good use of the Facial Action Coding System (FACS) (Ekman and Friesen 1978) for designing facially expressive automata, and there has been considerable work on computer-generated facial animation (Parke and Waters 1996).

Here we focus in particular on using procedural texture to convey some of the rich, time-varying facial expressions that people generally expect of each other. Aggressive, endearing, or otherwise idiosyncratic movements of facial expression convey a lot of our sense of another person's individuality. One inspiration for capturing this individuality can be found in successful animated characters.

For example, the hugely successful and endearing character of Gromit in Nick Park's animation (Park 1993) consistently reacts to events first with internal expressions of instinctive surprise or worry and then, a beat later, with some expression of higher judgment: disgust, realization, suspicion, and so on. Gromit's reactions become the audience's point of view. This identification creates an emotional payoff that draws in the audience (Park 1996). It would be good for an interactive character to be able to convey the same sense of a compelling emotional point of view, and to react with appropriate dynamic facial expression, without resorting to predefined expressions or to repetitive prebuilt animations.

To approach this with procedural textures, we build on Perlin and Goldberg (1996), using a parallel layered approach. The key is to allow the author of the animated agent to relate lower-level facial movements to higher-level moods and intentions, through controlled procedural textures. The animator specifies time-varying linear combinations and overlays of facial movements. Each such combination becomes a new, derived set of facial movements. In a later subsection we will show how to use controlled *noise* (Perlin 1985) to introduce controllable autonomous motion, and we will show a number of examples of autonomous facial animation built with this texturing approach.

Team LRN

Related Work

Facial movement synthesis by linear motion blending has been around for quite some time. In addition to Parke's work, this basic approach was also used by DeGraf and Wahrman (1988) to help drive an interactive puppeteered face.

Kalra et al. (1991) proposed an abstraction model for building facial animation, for combining vocal articulation with emotion, and for facilitating specification of dialogs in which characters could converse while conveying facial emotion. The model consisted of five abstraction layers:

1. Abstract muscles
2. Parameters built from these abstract muscles
3. Poses built by mixing parameters
4. Sequences and attack/sustain/release envelopes of poses
5. High-level scripts

Within this structure, the facial expression textures that we will describe largely contribute between levels 2 and 3 of Kalra's formalism, by providing a mechanism for combining multiple time-varying layers of successively abstracted facial expression.

Brooks (1986) developed autonomous robots having simultaneous different semantic levels of movement and control. In his *subsumption architecture,* longer-term goals were placed at higher levels, and immediate goals (such as "don't fall over") were placed at lower levels. When necessary, the lower-level controls could temporarily override higher-level activities.

Using the basic noise-based procedural texture tools of Perlin (1985), including *bias* and *gain* controls, I developed a responsive real-time dancer whose individual actions were procedurally defined (Perlin 1995). Actions could be layered and smoothly blended together to create convincing responsive body movement, which conveyed varying moods and attitudes. Athomas Goldberg and I later extended this work (Perlin and Goldberg 1996) to create characters that used a more advanced layering system for multiple levels of responsive motion. This continuous movement model was controlled by a discrete stochastic decision system, which allowed authors to create characters that would make choices based on dynamic moods, personalities, and social interactions, both with each other and with human participants.

Team LRN

The Movement Model

Assume that a face model consists of a set of vertices, together with some surface representation built on those vertices. We can build a component movement as a linear displacement of some subset of vertices. Thus, each movement is a list of $[i, \underline{x}]$ pairs, where i is a vertex identifier and $\underline{x}$ is the displacement vector for that vertex. We add additional movements for the axes of rigid head rotation, which are applied after all vertices have been displaced.

Given K component movements, we can give a state vector for the facial expression, consisting of linear combinations of its component movements. I used this approach to build a face that was as simple as possible, including only vertices that were absolutely necessary (Figure 12.28).

The goal was to make a model that would be used for working out a facial expression–texture vocabulary. Then the small number of vertices of this model could subsequently be used to drive movement of more elaborate facial geometries.

The face model used throughout this section contains fewer than 80 vertices, including the hair and shoulders. It requires no 3D graphical support and runs as an interactive Java applet on the Web (Perlin 1997).

The basic component movements included are

- Left/right eyebrows
- Left/right upper eyelids
- Left/right lower eyelids
- Horizontal/vertical eye gaze

FIGURE 12.28 The movement model.

- Left/right sneer
- Left/right smile
- Mouth open
- Mouth narrowed
- Head rotation

In this chapter we will refer to each of these component movements by the following respective names:

- `BROW_L, BROW_R`
- `WINK_L, WINK_R`
- `LLID_L, LLID_R`
- `EYES_R, EYES_UP`
- `SNEER_L, SNEER_R`
- `SMILE_L, SMILE_R`
- `AHH`
- `OOH`
- `TURN, NOD`

Internally, each component movement is represented by a *K*-dimensional basis vector, each of which has a value of 1 in some dimension *j* and a value of 0 in all other dimensions. For example:

```
BROW_L = (1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
BROW_R = (0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

The state space for the face consists of all linear combinations of these basis vectors. Commonly used combinations can be created rather easily. For example:

```
BROWS = BROW_L + BROW_R
BLINK = WINK_L + WINK_R
SNEER = SNEER_L + SNEER_R
```

Figure 12.29 shows some simple combinations of component movements. Figure 12.30 illustrates simple movement combinations and summation of component movements, first in wire frame and then shaded.

Each component movement also defines its opposite. For example, (−1 * `SMILE`) creates a frown. By convention, the face is modeled in a neutral expression, with the

Team LRN

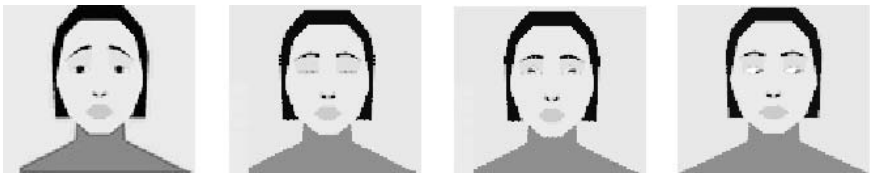


FIGURE 12.29 Simple component movement combinations.

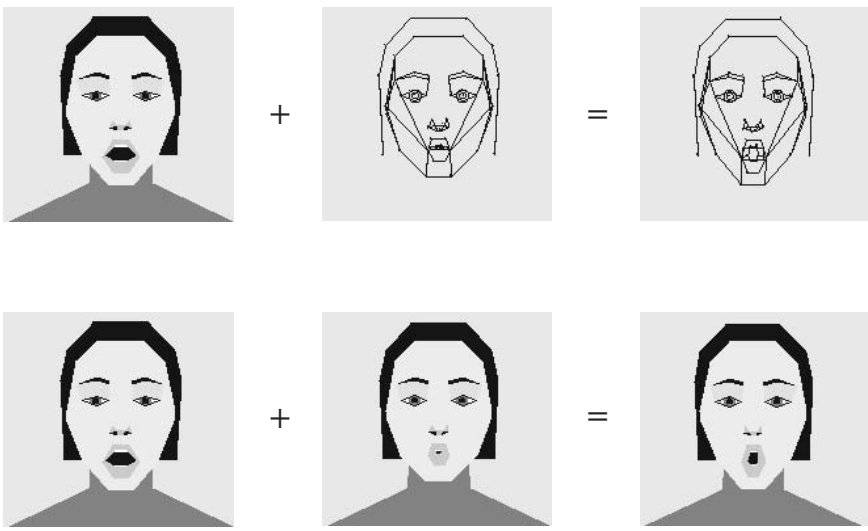


FIGURE 12.30 Summation of simple component movements in wire frame and shaded.

mouth half open, and the component movements are defined so that a range of −1 to +1 will produce a full range of natural-looking expressions.

Clearly, this is a simplified and incomplete model, sufficient just for the current purposes. The model is flexible enough to allow building textural facial expressions, but simple enough that it could be worked with quickly and easily for experiments. For experimental purposes, 16 component movements are just sufficient to allow an emotionally expressive face to be built.

Movements absent from this set include the ability to puff out or suck in the cheeks, to stick out or wave the tongue, to thrust the lower jaw forward or back, to tilt the head to the side, to displace the entire mouth sideways, as well as others. These should certainly be included in a complete facial model, which could either

Team LRN

retain our simple abstracted geometry or else drive a nonlinear muscle/skin model such as that of Lee, Redner, and Uselton (1985) or Chadwick, Haumann, and Parent (1989).

Movement Layering

The lowest-level abstraction is built on top of the primitive movements, using them as a vocabulary. We extend the model for mixing and layering that was described in Perlin (1995) and Perlin and Goldberg (1996). As in that model, movement is divided into *layers*. Each layer is a collection of related actions that compete for control of the same movements. Within a layer, one or more actions are always active. When a particular action A_i in a layer is activated, then the weight W_i for that action smoothly rises from zero up to one, and simultaneously the weights of all other actions in the layer descend smoothly down to zero, with a default transition time of one second.

Each action modulates the value of a set of movements (usually a fairly small set). Action A_i produces a mapping from *Time* to a list of *ComponentMovement, Value* pairs:

$$A_i : Time \text{ -> } ((D_1, V_1), \ldots, (D_k, V_k))$$

where each V_j is a scalar-valued time-varying function. In this way, an action influences one or more movements toward particular values. $A_i(Time)[D]$ refers to the value in action A_i at *Time* for component movement D.

Generally, more than one action is occurring within a layer. The total effect upon component movement D of all actions that influence it is given by the weighted sum

$$\sum_i A_i(Time)[D] * W_i(Time)$$

The total effect, or *opacity,* upon D from this group is given by

$$\sum_i W_i(Time)$$

All the groups run simultaneously. At any given moment, at least one action is running in each group. As in an optical compositing model, the groups are layered back to front. For each movement D, if the cumulative weight of a *Layer* at some *Time* is $opacity(Layer)(Time)$, then the results of all previous group influences on D are multiplied by $1 - opacity(Layer)(Time)$.

Team LRN

retain our simple abstracted geometry or else drive a nonlinear muscle/skin model such as that of Lee, Redner, and Uselton (1985) or Chadwick, Haumann, and Parent (1989).

Movement Layering

The lowest-level abstraction is built on top of the primitive movements, using them as a vocabulary. We extend the model for mixing and layering that was described in Perlin (1995) and Perlin and Goldberg (1996). As in that model, movement is divided into *layers*. Each layer is a collection of related actions that compete for control of the same movements. Within a layer, one or more actions are always active. When a particular action A_i in a layer is activated, then the weight W_i for that action smoothly rises from zero up to one, and simultaneously the weights of all other actions in the layer descend smoothly down to zero, with a default transition time of one second.

Each action modulates the value of a set of movements (usually a fairly small set). Action A_i produces a mapping from *Time* to a list of *ComponentMovement, Value* pairs:

$$A_i : Time \text{ -> } ((D_1, V_1), \ldots, (D_k, V_k))$$

where each V_j is a scalar-valued time-varying function. In this way, an action influences one or more movements toward particular values. $A_i(Time)[D]$ refers to the value in action A_i at *Time* for component movement D.

Generally, more than one action is occurring within a layer. The total effect upon component movement D of all actions that influence it is given by the weighted sum

$$\sum_i A_i(Time)[D] * W_i(Time)$$

The total effect, or *opacity,* upon D from this group is given by

$$\sum_i W_i(Time)$$

All the groups run simultaneously. At any given moment, at least one action is running in each group. As in an optical compositing model, the groups are layered back to front. For each movement D, if the cumulative weight of a *Layer* at some *Time* is $opacity(Layer)(Time)$, then the results of all previous group influences on D are multiplied by $1 - opacity(Layer)(Time)$.

Team LRN

The Bottom-Level Movement Vocabulary

At the bottom level are actions that simply pose the face or else make simple movements. This level has separate layers for head position, facial expression, and mouth position. The following are some actions in each layer. All are simple poses, except for `headroll`, `headbob`, and `headshake`. The latter two are controlled by *noise*.

From the head position layer:

- `action headback { (TURN, -0.2) ), (NOD, -0.5) }`
- `action headbob { (TURN, 0), (NOD, noise(2*Time)) }`
- `action headdown { N(NOD, 1) }`
- `action headroll { (TURN, cos(Time) ), (NOD, sin(Time)) }`
- `action headshake { (TURN, noise(2*Time)), (NOD, 0) }`

From the facial expression layer:

- `action angry { (BROWS, -1), (SMILE, -.8) }`
- `action distrust { (BROW_L, -1), (LLID_L, .5), (WINK_L, .5) }`
- `action nono { (AHH, -.6), (OOH, .6), (BROWS, -1), (BLINK, .1) }`
- `action sad { (AHH, -1), (BROWS, 1), (SMILE, -.8) }`
- `action smile { (BLINK, .4), (LLIDS, .5), (BROWS, 0), (SMILE, 1), (OOH, -.6) }`
- `action sneeze { (AHH, -1), (BLINK, 1), (BROWS, 1), (LLIDS, .7), (SNEER, .7) }`
- `action surprised { (AHH, .1), (BROWS, 1), (BLINK, -.5), (LLIDS, 1) }`
- `action suspicious { (AHH, -.9), (BLINK, .5), (BROW_R, -1.2), (BROW_L, -.5), (LLIDS, 1.1) }`

From the mouth position layer:

- `action say_a { (AHH, .1), (OOH, 0) }`
- `action say_e { (AHH, -.6), (OOH, -.2) }`
- `action say_f { (AHH, -.9), (OOH, -.2), (SNEER, .2) }`
- `action say_o { (AHH, -.1), (OOH, .3) }`
- `action say_r { (AHH, -.6), (OOH, .2) }`

Team LRN

- `action say_u { (AHH, -.6), (OOH, .6) }`
- `action say_y { (AHH, -.7), (OOH, -.3) }`

Figure 12.31 shows the results of some pose actions.

Painting with Actions

Up to this point, the model mimics that of Perlin and Goldberg (1996), by allowing simple layered combinations of the primitive movements. It would be useful to go beyond this, treating an action as a texture. For example, the actions defined earlier can place the face into useful positions, and the layered blending mechanism will make smooth transitions between those positions. Yet the face will appear to move in a fairly lifeless and mechanical way.

To improve on this, we would like to mix some coherent jitter into many component movements, tuned to match the subtle shifting that real faces do. This is done first by defining an action that creates the jitter and then by "mixing in" amounts of this action as a transparent wash on top of the preexisting motion.

More generally, an action is a time-varying function of some set of movements. This action itself can be viewed as a primitive, which can be added and mixed. Consider the analogy with painting. An artist begins with a set of paints, each having a discrete color. The artist then creates a palette by mixing these colors to get new colors. The artist can then use each of these new colors without needing to go back to the original color set. In digital paint systems, the artist can do this with textures as well. The artist can create a texture and then paint with it, as though painting with a custom-designed textured brush.

Similarly, an action provides a way to encapsulate a time-varying function of movements and to treat this time-varying function as though it were itself a primitive movement. If noise-based variation in the action definition creates some motion "texture," then any uses of this action will reflect that texture.

Here is a more sophisticated example, which contains time-varying behavior:

```
action sneeze {
    Ah = timeCurve((0,0),(1,1),(1.5,0));
    Choo = timeCurve((1.2,0),(1.5,1),(2,0));
    ( sneeze, Ah),
    ( headup, Ah / 3),
    ( nono, Choo),
    ( headshake, Choo / 2),
    ( headdown, Choo / 2)
}
```

Team LRN

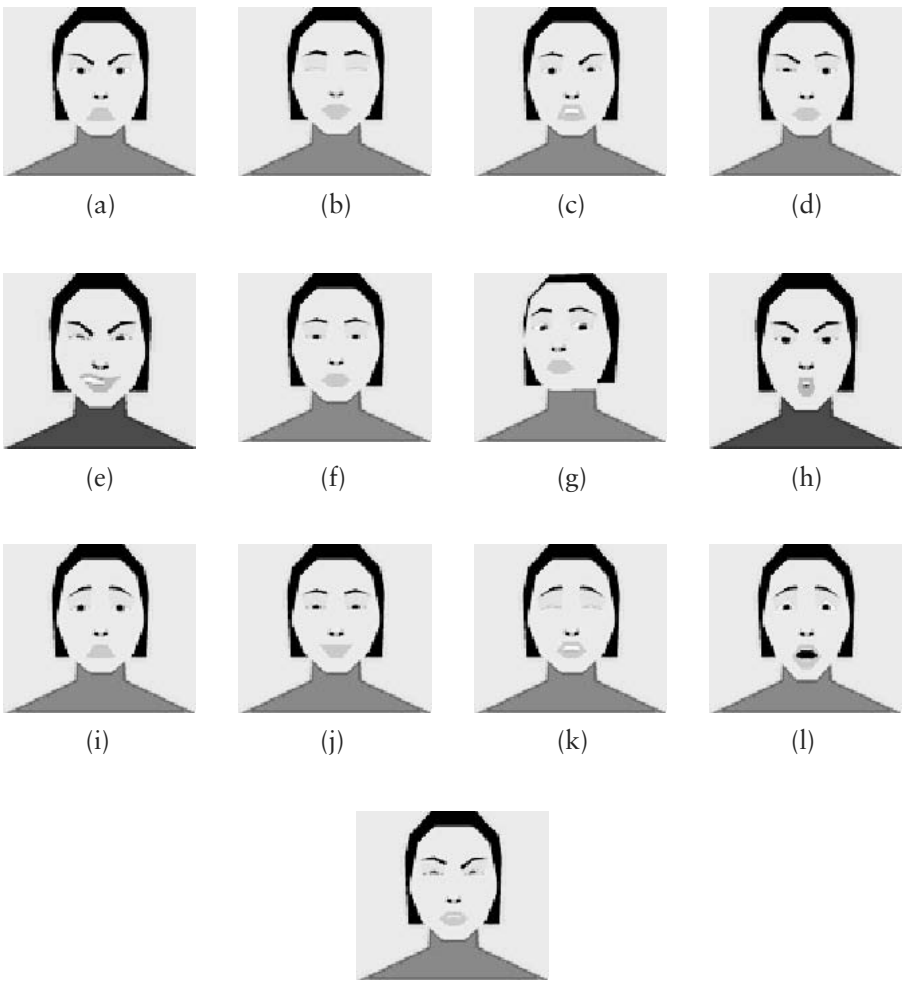


FIGURE 12.31 Results of pose actions: (a) angry; (b) daydreaming; (c) disgusted; (d) distrustful; (e) fiendish; (f) haughty; (g) head back; (h) disapproving; (i) sad; (j) smiling; (k) sneezing; (l) surprised; (m) suspicious.

Team LRN

where `timeCurve(...)` is a function that interpolates a smooth spline between a given sequence of `(time,value)` pairs, based on elapsed time since the onset of the action.

The animator does not need to know the details of the lower-level definitions for `sneeze`, `headup`, and so on, which frees the animator to concentrate on the details that are important at this level: tuning the time-varying behavior of the sneeze. Note that in this example the derived action inherits, via `headshake`, a realistic nondeterministic shaking during the `Choo` phase of the sneeze. The animator gets this nondeterminism for free.

Using *noise* in Movement

A number of examples will illustrate the use of *noise* in layering facial movement.

Blinking

Noise is used to trigger blinking with controlled irregularity. After each blink, we wait for one second. Then we start to check the value of a time-varying *noise*, which ranges smoothly from −1.0 to 1.0. When the value of this *noise* exceeds 0.1, we trigger another blink:

```
action blink {
  if (Time X blink.Time + 1 and noise(Time) X 0.1)
    blink.Time = Time
    (BLINK, 1) // set BLINK on this frame
  }
```

Small Constant Head Movements

As we said earlier, we would like to mix some coherent *noise* into the movement, tuned to match the subtle shifting that real faces do. To do this in a way that works with all other movement, we overlay a transparent layer of random movement over the activities of the bottom abstraction. First we add an action into the bottom abstraction that creates random movements:

```
action jitter {
    T = noise(Time);
  add (BROWS, T);
  add (LLIDS, T);
  add (EYES_R, T/2);
  add (EYES_UP, T/2);
  add (TURN, noise(Time + 10));
  add (NOD, noise(Time + 20));
}
```

Team LRN

Then we add an action to the next level of abstraction that creates a transparent mix-in of the lower-level jitter:

```
action jitter { jitter 0.3 }
```

The effect is subtle, yet essential. It gives a sense of life and movement to whatever the face is doing.

Both blinking and jitter are controlled by *noise*. This allows us to make movements that are unpredictable to the observer and yet that have tunable statistics that match users' expectations.

Simulated Talking

We found that we could also use tuned *noise* to create a fairly realistic simulation of the mouth movements a person makes while talking. This would be useful, for example, when an agent or character is pretending to hold a conversation off in the background:

```
action talking {
  T = (1 + noise(2 * Time)) / 2; // noise between 0 and 1

    add (AHH, lerp(gain(.9, bias(.3, T)), -1, .8)); // snap open/closed
  add (OOH, noise(2 * Time + 10.5)); // vary smoothly
  add (SNEER, lerp(bias(T, .1), -.1, .6)); // the "f" phoneme
}
```

In this implementation, we first define a *noise* process that has a frequency of two beats per second, with values in the range 0 to 1. We also use *bias*() and *gain*() as defined in Perlin (1985) to shape the *noise,* as well as linear interpolation, defined by

$$lerp(t, a, b) = a + t(b - a)$$

We use these tools to do several things simultaneously:

- Snap the mouth open and closed fairly crisply (the `gain(.9, bias(.3, T))` takes care of this).
- Smoothly make the mouth wider and narrower.
- Occasionally show the upper teeth via a `SNEER`. This gives the visual impression of an occasional *f* or *v* sound.

Once the parameters have been tuned, the result is surprisingly convincing. We use a derived abstraction to smoothly turn this talking action on and off, as when two characters are taking turns during a conversation.

Team LRN

Searching

We can also create a lifelike motion to make the agent appear to be addressing an audience. When people do this, they tend to look at particular people in their audience in succession, generally holding each person's gaze for a beat before moving on. We can simulate this with *noise,* by making the head and gaze follow a controlled random moving target. First we create a target that darts smoothly around, lingering for a moment in each place:

```
T = Time/3 + .3 * sin(2*PI * Time/3);   // make time undulate
Target = (noise(T), noise(T + 10) / 2); // choose moving target
```

Then we direct the `TURN`, `NOD`, `EYES_R`, and `EYES_UP` movements to follow this target point.

The first line in the previous code undulates the domain of the *noise* so that the target point will linger at each location. This is a special case of expressions having the form $X/S + A * \sin(2*\pi*F/S)$, with A in the range [0..1], which creates a monotonically increasing function that undulates regularly between slow and fast movement, with an average slope of $1/S$. In particular, the function becomes slow every S units on its domain.

By feeding such an undulating function into *noise,* we have specified a smoothly moving point that slows to a near stop at a different random target point every three seconds. We have found this to be quite effective together with the talking simulation, to create a realistic impression of a person addressing an audience.

Laughing

Here we create a laugh motion that starts by rearing the head up momentarily, and then settles into a steady state of bobbing and shaking the head while modulating an open *o* sound with a *noise* function:

```
action laugh {
  ( headshake , 0.3 ),
  ( headbob , 0.3 ),
  ( headup , timeCurve( (0,0) , (.5,.5) , (1,0) ) ),
  ( smile , 1 ),
  ( say_o , 0.8 + noise(Time) )
}
```

We can then mix this in, as a fleeting action, by deriving a higher-level laugh. This more abstracted laugh leaves the face in persistent smiling mood, until some other response causes the laugh action to be turned off:

Team LRN

```
action laugh {
  ( laugh , timeCurve( (0,0) , (.5,1) , (2,0) ) ),
  ( smile , 1 )
}
```

Same Action in Different Abstractions

We now describe several more actions that have been implemented at multiple semantic levels, within successively derived abstractions.

Wink

At the lowest level, we can express a wink as just the closing of one eye:

```
action wink { ( WINK_L , 1 ) }
```

At the next level we can mix in a sly or distrustful expression with the head slightly turned to one side:

```
action wink {
H = timeCurve( (0,0),(.5,.4),(.6,.5),(.9,.5),(1,.4),(3,0) );
W = gain(.999, bias(.1, 2 * H));
( wink , W ),
( distrust , H ),
( headright , 0.2 * H )
}
```

This has consistent but different effects when it is invoked while the face is in different moods: smiling, haughty, distrustful (Figure 12.32).

Note also that these haughty and distrustful facial expressions are more effective than were their lower-level equivalents in Figure 12.29. At this higher level of abstraction they have been combined with the most effective head positions. Because head position and facial expression have been mixed within a higher abstraction layer, each is correctly modulated by the wink.

Talking in Different Moods

Finally, we have created various combinations of the autonomous talking process overlaid with different moods. The movements of the face can be seen interactively at *mrl.nyu.edu/perlin/facedemo*.

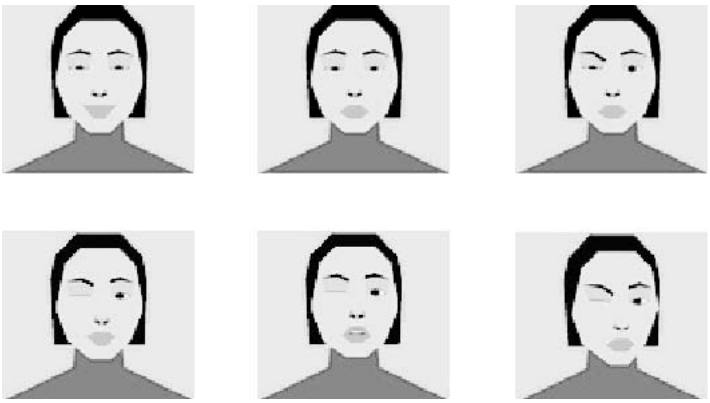


FIGURE 12.32 Various moods.

What Next?

Currently, the interface for creating individual actions is entirely script based, although I use simple drawing widgets for defining time-varying splines. There is ongoing work to create a graphic user interface, so that animators can move sliders interactively to modify component movements and mix-in weights.

Beyond this, I'm planning to extend the paint-mixing metaphor for blending actions. Within any abstraction, the animator will see a palette of small animated faces that represent the primitives available to that abstraction. The animator can then blend these to modify an action, using a time-varying brush.

I'm also continuing to increase the vocabulary of movements, working mostly from direct observation. For example, one movement to add is amused disbelief, which combines eyes looking up, smiling, and head shaking from side to side. Another is "You want this, don't you?" which combines head turned to the side, eyes looking at you, eyes wide upon, and eyebrows up. A third is puzzlement, which combines head tilted to the side (requiring an additional degree of freedom), the lower lids up, and one eyebrow raised.

In near future work, I'd like to categorize idioms that consist of mixing of lower and higher levels. One goal is to implement a Gromit-like character by the use of these mixed-level idioms. For example, the character can raise one eyebrow and then shake its head a beat later. A test of this would be to attempt to implement some of Gromit's basic emotional reaction vocabulary.

Team LRN

CONCLUSION

Since much of this chapter was first written, a huge revolution in procedural texturing and modeling has been taking place. Just in the last year, we have seen enormous strides in the capabilities and programmability of graphics accelerator boards. This coming SIGGRAPH (2002) will mark the first year that graphics accelerator companies will be making truly general-purpose graphics hardware–level programming available to high-level ("C style") programmers, so that their algorithms can be compiled down to accelerated and highly parallel pixel shaders.

This year, these hardware pixel shader capabilities still provide only SIMD (single instruction, multiple data) parallelism, which means that many of the techniques I began to teach about in 1984, which make much use of data-based branching and looping, are not yet available down at this fastest level.

But the gauntlet has been flung. Each year in this decade is going to see great leaps on this hardware accelerated level of capability, and at some point in the next few years we'll see high-level programmable MIMD (multiple instruction, multiple data) parallelism. At that point we will truly have real-time procedural texturing and modeling, and the world I was trying to give people a glimpse into 18 years ago will finally have arrived—a world, in the immortal words of Lance Williams, "limited only by your imagination."

Team LRN

13

REAL-TIME PROCEDURAL SOLID TEXTURING

JOHN C. HART

As much of this book has demonstrated, procedural solid texturing is a powerful tool for production-quality image synthesis. Procedural solid textures can allow users to explore worlds flush with nonrepeating mountains, coastlines, and clouds. Dynamic animated textures like fire and explosions can be represented efficiently as procedural textures. Objects sculpted from a textured medium, like wood or stone, can be textured using solid texturing. Solid texturing is also easier than surface texturing because it does not require a surface parameterization.

However, much of the benefit of procedural solid texturing has been limited to the offline rendering of production-quality scenes. With the advent of programmable shading hardware in modern graphics accelerators, procedural solid texturing is becoming a useful tool for real-time graphics elements in video games and virtual environments. Procedural solid textures are compact and can be synthesized dynamically on demand. They can provide video games and virtual environments with a vast variety of textures that require little additional storage, at a resolution limited only by machine precision.

This chapter describes how to integrate procedural solid texturing into real-time systems using features already available in current graphics programming libraries. These techniques can also be used to create an interactive procedural solid texturing design system with real-time feedback.

A REAL-TIME PROCEDURAL SOLID TEXTURING ALGORITHM

The real-time procedural solid texturing algorithm is based on a technique from RenderMan that allows the texture map to hold the shading of a surface (Apodaca 1999). We assume each vertex in our model is assigned three kinds of coordinates. The *spatial coordinates* x, y, z of the vertex describe the location of the vertex in model space. The *parameterization* u, v of the vertex describes the location of the vertex in a 2D texture map. The *solid texture coordinates* s, t, r of the vertex describe

Team LRN

from where in a 3D procedural texture space the vertex should get its color or other shading information. Given the spatial coordinates and the solid texture coordinates, we will construct a parameterization automatically. Often the solid texture coordinates are simply set equal to the spatial coordinates, so this algorithm can texture an object given only its spatial texture coordinates. The algorithm will run in three phases: rasterization, procedural evaluation, and texture mapping (see Figure 13.1). There is also a preprocessing step that we will call atlas construction that assigns the texture coordinates u, v to each vertex. This step will be described in the next section.

The rasterization phase plots the object's polygons in the texture map. The parameterization u, v serves as the coordinates of the vertices, and the solid texture coordinates serve as the color ($R = s$, $G = t$, $B = r$) of the vertices. Graphics hardware has long supported the linear interpolation of attributers across a polygonal face,[1] needed, for example, for smooth Gouraud shading. This linear interpolation automatically calculates the solid texture coordinates across the face of the polygon as it is rasterized in the texture map.

Rasterization

```
For each polygon p
        Begin polygon
        For each vertex i
                Color(p[i].str)
                Vertex(p[i].uv)
        End polygon
Save image as texture map tex
```

The procedural evaluation phase marches through all of the pixels in the texture map and evaluates the texturing procedure on the solid texture coordinates (s, t, r) stored as the pixel's RGB color. This evaluation will result in a new RGB color that represents the texture at the point (s, t, r) in the solid texture space. This color is stored at the current pixel in the texture map, overwriting the solid texture coordinate with its corresponding procedural texture color.

Procedural Evaluation

```
For each pixel (x, y) in the texture map tex
        tex[x, y] = proc(tex[x, y])
```

1. This interpolation is actually projective such that texture is interpolated "perspective correct."

Team LRN

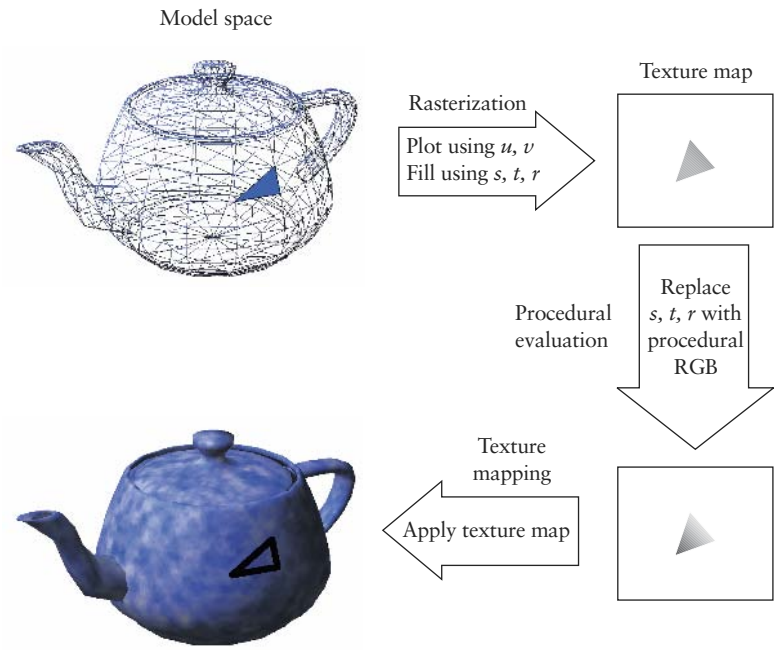


FIGURE 13.1 An algorithm for procedural solid texturing.

The texture mapping phase places the texture back on the object via standard texture mapping using the object's *u*, *v* parameterization. When the object is drawn, the spatial coordinates of its polygon vertices are passed through the graphics pipeline. The polygon is rasterized, which interpolates its *u*, *v* parameterization and textures the polygon using the corresponding pixel from the procedural solid texture stored in the texture map.

Texture Mapping
Set texture map to *tex*
For each polygon *p*
    Begin polygon
    For each vertex *i*
        TexCoord(*p*[*i*].*uv*)
        Vertex(*p*[*i*].*xyz*)
    End polygon

Team LRN

For this technique to work, the polygons on the object surface need to be laid out in the texture map without overlap. Such a texture mapping is called an *atlas*. The atlas construction step is performed as a preprocess to the previous algorithm and only needs to be computed once for an object. Techniques for performing this layout are described in the next section.

Models usually need one or more cuts in order to be laid flat into a texture map. These cuts can result in seams, which appear as discontinuities in the texture. Some texture layout techniques have focused on reducing the number and length of these seams. The section "Avoiding Seam Artifacts" shows how seams can be avoided for real-time procedural solid texturing, which allows the layout methods described in the next section to ignore seam length altogether and pack triangles individually into the texture atlas.

Both the rasterization and the texture mapping steps are hardware accelerated. The procedural evaluation step remains the bottleneck. Later in this chapter we will describe some techniques for efficient implementation of procedural textures, in particular those based on the Perlin *noise* function.

CREATING AN ATLAS FOR PROCEDURAL SOLID TEXTURING

A variety of techniques have been developed for creating texture atlases. Some of these techniques have been developed to automatically parameterize an object in order to place a 2D texture on its surface. These techniques try to minimize distortion, such that the proportions of the texture map image are reasonably reproduced on the textured surface. Because the real-time procedural solid texturing algorithm computed the texture from solid texture coordinates stored per pixel in the texture map, the distortion of the atlas does not affect the proportions of the procedural solid texture. The scaling component of the distortion, quantified in various forms elsewhere as the *stretch* (Sander et al. 2001) or the *relative scale* (Carr and Hart 2002), can affect the distribution of the samples across the object surface.

Because rasterization fills in the pixels in the texture map with solid texture coordinates, the location of triangles relative to each other in the texture map can be made irrelevant. Hence neighboring triangles in the object need not be neighbors in the texture map. This greatly simplifies the process of laying out the triangles in the texture atlas.

In order to use as many of the available texture pixels as possible, we lay out the triangles in a mesh across the entire texture map. This mesh consists of rows of isosceles axis-aligned right triangles that pack in a very straightforward manner, as shown in Figure 13.2.

Team LRN

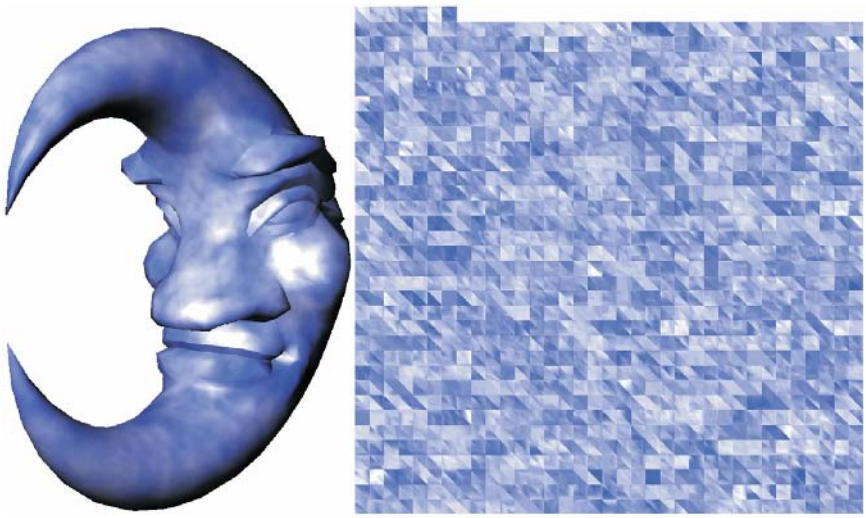


FIGURE 13.2 A uniform meshed atlas.

Laying out all object triangles into uniformly sized texture map triangles does not distribute texture samples well. Large object triangles should have more procedural solid texture samples than smaller object triangles, but the uniform mesh atlas assigns them the same number of samples. This can result in blocky texture artifacts on larger triangles and in general wastes texture space by giving smaller triangles too many texture samples.

We can also adjust the uniform mesh atlas to distribute the available texture samples more evenly across the object surface by varying the size of triangles per strip, as shown in Figure 13.3. We strip-pack the triangles into the texture map in order of nonincreasing area. We estimate a uniform scale factor as the ratio of the surface area to the texture area and use this scale factor to set the size of triangles in each horizontal strip. All of the texture map triangles in each horizontal strip are set to the same size to avoid wasting texture samples.

Other techniques have been developed to pack triangles of different sizes into a texture atlas. Maruya (1995) treated the triangles of a uniform mesh atlas as blocks and packed these blocks with triangles of a variety of sizes quantized to the nearest power of two. Rioux, Soucy, and Godin (1996) paired triangles of similar sizes into square blocks (quantized to the nearest power of two) and packed these blocks into the texture map. Battke, Stalling, and Aege (1996) rigidly strip-packed the triangles into the texture map, scaling them all uniformly so they would fit. Cignoni et al.

Team LRN

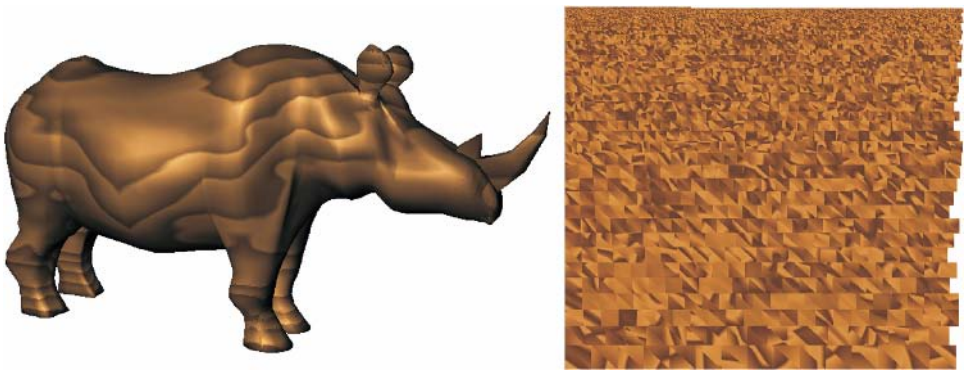


FIGURE 13.3 Rhino sculpted from wood and its area-weighted mesh atlas.

(1998) performed a similar strip packing, but sheared the triangles to better fit together.

Figure 13.3 shows an example of a shape shaded with a procedural solid texture using an area-weighted mesh atlas. The per-strip sizing of the triangles provides a more even distribution of texture samples across the surface than the other area-weighted layouts that quantize to the nearest power of two. Those techniques tend to more easily fill the texture map, whereas the area-weighted mesh has a blocky edge of wasted texture samples.

Skinny triangles can confuse the area-weighted atlas. A skinny triangle has two long edges that should receive more samples, but its surface area can be arbitrarily small. Meshes with a significant number of skinny triangles can still exhibit some texture blockiness using the area-weighted atlas.

For these cases, a length-weighted atlas is a better approach (Rioux, Soucy, and Godin 1996). Instead of using the triangle's surface area to set the size of its image in the texture map, the length-weighted approach uses the length of the triangle's longest edge. This technique ensures that the longest edges in the mesh get the most samples, but the technique tends to oversample skinny triangles, and this waste reduces the number of samples available to other areas of the model.

Recent atlases have also been designed to support MIP mapping. An atlas by Sander et al. (2001) clusters regions of triangles and packs these clusters into the atlas. The regions between clusters are filled with "reasonable" colors that allow it to be MIP mapped. An atlas by Carr and Hart (2002) uses a subset property of MIP mapping to pack the triangles into a hierarchy of proximate, although not necessarily neighboring, regions. Figure 13.4 shows an example of the proximate MIP map,

Team LRN

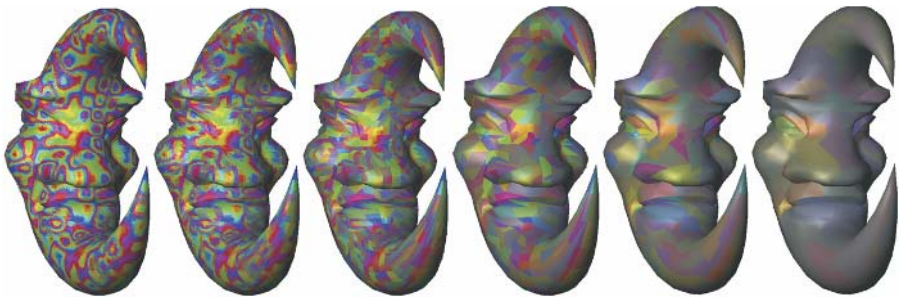


FIGURE 13.4 MIP mapping based on clusters of proximate triangles.

which maps proximate clusters of mesh triangles into the same quadrant at a given MIP map level.

AVOIDING SEAM ARTIFACTS

Texture filtering can cause some samples near the boundary of the polygon to be drawn from the wrong polygon image in the texture map. If the triangle is a neighbor, then this error is not at all serious. But since we are ignoring polygonal neighborhoods, we cannot depend on this situation.

Consider the example in Figure 13.5. In this example, the texture is a 4×4 pixel square. We will use integer coordinates for the pixel centers, with the lower-left pixel at (0,0). Two triangles are rasterized. The darker triangle has coordinates $(-0.5, -0.5)$, $(4.5, -0.5)$, and $(-0.5, 4.5)$, and the lighter triangle has coordinates $(-0.5, 4.5)$, $(4.5, -0.5)$, and $(4.5, 4.5)$.

The pixels in Figure 13.5(a) are assigned according to the rules of rasterization. These rules were designed to avoid competition over pixels that might be shared by multiple polygons. Pixels that fall in the interior of the polygon are assigned to the polygon. Pixels that fall on the shared edge of a pair of polygons are assigned to the color of the polygon on the right. If the shared edge is horizontal, they are assigned to the color of the polygon above. Hence the pixels that fall along the hypotenuse are assigned to the lighter triangle.

The two triangles in Figure 13.5(a) delineate two sampling regions. During texture-mapped rasterization, interpolated u, v coordinates may fall within the bounds of either of these two texture map triangles. These coordinates will likely not lie precisely at pixel centers. They instead need to sample a continuous function reconstructed from the discrete pixel samples by a reconstruction filter.

Team LRN

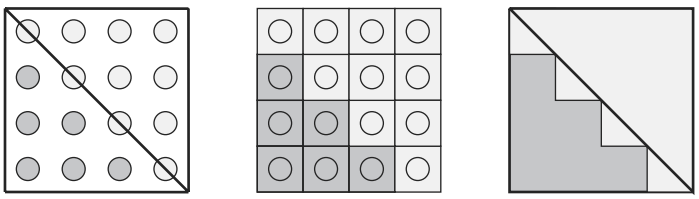


FIGURE 13.5 (a) Textures are stored as an array of pixels. (b) The nearest-neighbor filter returns the color of the closest pixel to the sampled location. (c) This can result in seam artifacts, shown here as the light triangle color bleeding into the darker triangle's sampling region.

The nearest-neighbor filter is the simplest such reconstruction filter, so we will begin our analysis with it. Nearest neighbor returns the color of the pixel closest to the sample location. Because the texture is stored in a rectilinear grid of pixels, nearest-neighbor sampling surrounds each of the pixel centers with square Voronoi cells of constant color. Hence the nearest-neighbor filter reconstructs from the discrete pixels in Figure 13.5(a) a continuous function consisting of square regions of constant color as illustrated in Figure 13.5(b).

Seams appear in part because the rules of texture filtering are inconsistent with the rules of rasterization. Samples taken anywhere within the lighter triangle draw from an appropriate nearest-neighbor pixel. But some positions in the darker triangle near the hypotenuse have, as their nearest neighbor, pixels assigned by rasterization to the lighter triangle. This results in the staircased seam in the sampling region of the darker triangle. The two triangles are unrelated and could appear in completely different locations on the object surface, representing completely unrelated textures.

We need to lay out the triangles in the texture map so they can be sampled correctly. The solution to this problem is to offset the triangles sharing the hypotenuse by one pixel horizontally (Rioux, Soucy, and Godin 1996), as shown in Figure 13.6. We rasterize an enlarged darker triangle with vertices $(-0.5, -0.5)$, $(5.5, -0.5)$, and $(-0.5, 5.5)$ and a translated lighter triangle with vertices $(0.5, 4.5)$, $(5.5, -0.5)$, and $(5.5, 4.5)$ as shown in Figure 13.6(a). The rasterization of these new triangles shades the pixels as shown in Figure 13.6(b), giving an equal number of pixels to each triangle. The texture coordinates u, v of the darker triangle have not changed, and so the smaller triangle is sampled properly from the overscanned rasterization as shown in Figure 13.6(c). The lighter triangle is similarly sampled from its translated position.

Bilinear filtering can also be supported (Carr and Hart 2002) by using the sampling regions shown in Figure 13.6(d). In this case, triangles that share a hypotenuse

Team LRN

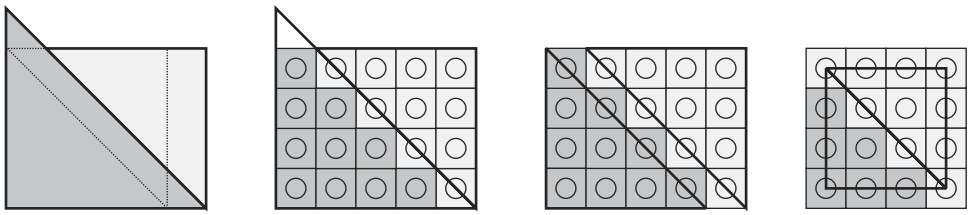


FIGURE 13.6 Avoiding seam artifacts.

in the atlas must share an edge in the mesh. The vertices of the sampling region have been inset one-half pixel, providing the buffer necessary to support bilinear filtering.

IMPLEMENTING REAL-TIME TEXTURING PROCEDURES

The procedural evaluation step of our real-time procedural solid texturing algorithm executes the texturing procedure on the texture atlas after the rasterization step has filled it with solid texture coordinates. The procedural evaluation step replaces the solid texture coordinates stored in the RGB values with a resulting texture color, which is applied to the object in the texture mapping step. This section explores various techniques for implementing fast texturing procedures. These techniques focus on implementations of the Perlin *noise* function (Perlin 1985), which is a common element found in many procedural textures.

One option is to implement the texturing procedure on the CPU. Several have implemented the Perlin *noise* function using special streamlined instructions available on Intel processors (Goehring and Gerlitz 1997; Hart et al. 1999). Our streaming SIMD implementation was able to run at 10 Hz on an 800 MHz Pentium III. The main drawback to CPU implementation is the asymmetry of the AGP graphics bus found in personal computers, which is designed for high-speed transmission from the host to the graphics card but not vice versa. In fact, we found that when we used the CPU to perform the procedure evaluation step, it was faster to also perform the atlas rasterization step on the CPU instead of the GPU because the result of the CPU implementation did not require a readback through the AGP bus.

However, we should take advantage of the power of the graphics accelerator. This means we should take advantage of the programmable shading features available on modern GPUs in the implementation of the procedure evaluation step. Doing so also allows us to take advantage of the GPU during the rasterization step.

Team LRN

A variety of implementations exist using different components of modern graphics accelerators. The *noise* function can be implemented using a 3D texture of random values with a linear reconstruction filter (Mine and Neyret 1999). A texture atlas of solid texture coordinates can be replaced with these noise samples using the OpenGL pixel texture extension or dependent texturing. Others have implemented the Perlin *noise* function as a vertex program (NVIDIA 2001), but a per-vertex procedural texture produces vertex colors or other attributes that are Gouraud interpolated across faces. Hence the frequency of the noise is limited by the frequency of the tessellation.

The Perlin *noise* function can also be implemented as a multipass pixel shader (Hart 2001). This implementation is based on the formulation of the Perlin *noise* function as a 3D integer lattice of uniformly distributed random values from 0 to 1 (see Figure 13.7). These discrete lattice values are reconstructed into a continuous function through interpolation, which locally correlates the random values. Depending on the application, this reconstruction can be C^1 smooth, using cubic interpolation, or fast, using linear interpolation.

The multipass pixel shader implementation of the Perlin *noise* function is based on the Rayshade implementation (Skinner and Kolb 1991). That implementation of the *noise* function uses 3D reconstruction filter kernels at the integer lattice points, with amplitudes set to the lattice point random values

$$\sum_{k=0}^{1}\sum_{j=0}^{1}\sum_{i=0}^{1}\text{Hash3d}(\lfloor s\rfloor+i,\lfloor t\rfloor+j,\lfloor r\rfloor+k)w(s,i)w(t,j)w(r,k)$$

The summation iterates over all eight corners of the cube containing the point s, t, r. For each of these corners, the function Hash3d(x, y, z) constructs a random

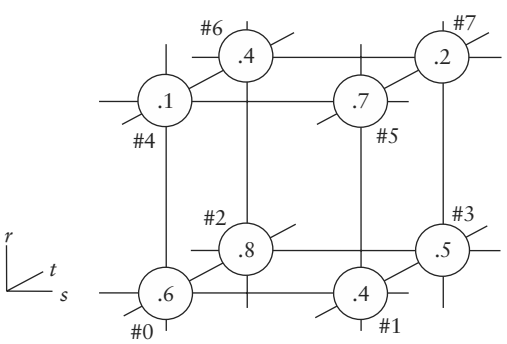


FIGURE 13.7 Noise based on an integer lattice of random values. Corner are labeled from #0 to #7 for later reference.

Team LRN

(a) (b) (c)

FIGURE 13.8 (a) A texture map of solid texture coordinates (*s*, *t*, *r*) ranging from (0, 0, 0) to (1, 1, 0), decomposed into (b) an integer part `atlas_int` and (c) a fractional part `weight`.

value t the integer lattice point *x*, *y*, *z* by performing an arbitrary set of bitwise operations on the integer coordinate values *x*, *y*, and *z*.

The Rayshade implementation of the *noise* function can be adapted to a multipass pixel shader. The basic outline of the multipass pixel shader implementation of *noise* is as follows.

1. The algorithm begins with the input RGB texture named `atlas` whose colors contain the interpolated *s*, *t*, *r* coordinates from the rasterization step (Figure 13.8(a)).
2. Initialize an output luminance texture `noise` to black.
3. Let `atlas_int` be an RGB texture whose pixels are the integer part of the corresponding pixels in `atlas`. Each pixel of `atlas_int` contains the coordinates of the lower-left front corner of the noise lattice cell that contains the coordinates of the corresponding pixel in `atlas` (Figure 13.8(b)).
4. Add the value one to each of the RGB components of the pixels of texture `atlas_int` to get the texture `atlas_int++`. Each pixel of `atlas_int++` now contains the coordinates of the upper-right back corner of the noise lattice cell that contains the coordinates of that pixel in `atlas`. Note that now all eight corners of the noise lattice cell can be constructed as a combination of the components of `atlas_int` and `atlas_int++`.
5. Let `weight` be a texture whose pixels are the fractional parts of the corresponding pixels in `atlas` (Figure 13.8(c)).

Team LRN

6. For $k = 0..7$:
   a. Let `corner` be an RGB texture equal to texture `atlas_int` overwritten with the texture `atlas_int++` using the color mask (k&1, k&2, k&4). The texture `corner` now contains the integer coordinates of corner #k of the cell.
   b. Let `random` be a luminance texture whose pixels are uniformly distributed random values indexed by the corresponding pixels of `corner`. The texture `random` now holds the noise value at corner #k (Figure 13.9(a)).
   c. Multiply `random` by the red channel of `weight` if (k&1), otherwise by one minus the red channel of `weight` (Figure 13.9(b)).
   d. Multiply `random` by the green channel of `weight` if (k&2), otherwise by one minus the green channel of `weight` (Figure 13.9(c)).
   e. Multiply `random` by the blue channel of `weight` if (k&4), otherwise by one minus the blue channel of `weight`. These three instructions have now computed the contribution of that corner's value in the trilinear interpolation (Figure 13.9(d)).
   f. Add `random` to `noise`.
7. At this point the pixels of the luminance texture `noise` will now contain values linearly interpolated from random values indexed by the coordinates of its eight surrounding cell corners (Figure 13.10).

Our first implementations of this algorithm were on pixel shaders that only allowed 8 bits of precision (Hart 2001). These implementations used fixed-point

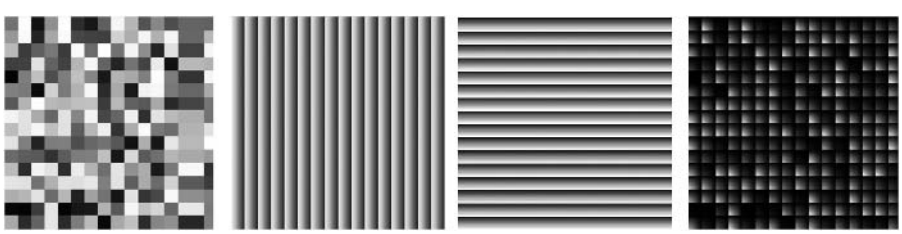


FIGURE 13.9 (a) Random values computed from `atlas_int`, weighted by (b) 1 − R(`weight`), and (c) 1 − G(`weight`) (and 1 − B(`weight`), which is uniformly equal to one), to produce (d) corner #0.

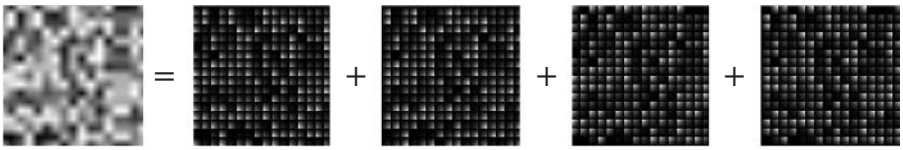


FIGURE 13.10 Noise is equal to the sum of corners #0 through #3. (Corners #4 though #7 are black since solid texture coordinate *r* is an integer, specifically zero, throughout this test.)

numbers with 4 bits of integer and 4 bits of fractional parts, and special pixel shader routines were developed for shifting values left or right to obtain the integer and fractional parts. Our original implementation also implemented a random number generator in the pixel shader, although modern graphics hardware now supports dependent texturing, which allows the `random` texture to be generated by using the components of the `corner` texture to index into a precomputed texture of random values.

APPLICATIONS

The real-time procedural solid texturing method is view independent. Hence, once the procedural solid texture has been computed on the atlas of an object, then the object can be viewed in real time. Each new view of the procedural solid texture on the object requires only a simple texture-mapped rendering, which is supported by modern graphics processors.

The most expensive operation of the real-time procedural solid texturing process is the generation of the atlas. Fortunately, the atlas need only be generated once per object. The atlas needs to be regenerated when the object changes shape. However, if the object deforms by changing only its vertex positions, then the atlas can remain unchanged (although the relative sizes of triangles on the object surface may have changed, which can result in a poor distribution of texture samples). Furthermore, if the object deforms but retains the same mesh structure, then the procedural solid texture will adhere to the object, overcoming the problem where the object swims through a solid texture, as shown in Figure 13.11.

The atlas encapsulates a procedural solid texturing of an object. The textured atlas can be attached to the object model as a simple 2D texture map, which is already supported by numerous object file formats.

The next most expensive operation is the procedural texture evaluation step mentioned earlier. The rasterization step can be performed on the graphics hardware, but we found it was sometimes useful to perform this step on the CPU given

(a) (b) (c)

FIGURE 13.11 (a) A plank of wood. (b) A curved plank carved out of wood versus (c) a warped plank of wood. Image courtesy of Jerome Maillot, Alias|Wavefront.

the constraints of current graphics hardware. The first reason was that it allowed the CPU to apply the procedural texture without performing an expensive readback of the rasterized pixels. The second reason was that it gave us full control over the rules of rasterization, which can sometimes vary between hardware implementations. As readback rates improve and graphics hardware becomes more programmable, implementation of all three steps on the GPU will certainly be the better practical choice.

We have used these techniques to construct an interactive procedural solid texturing design system. This system allows the procedural solid texturing to be manipulated via parameter sliders. As the sliders move, the resulting procedural solid texture is reapplied to the object. Since the shape of the object is not changing, the atlas does not need to be recomputed, but the procedural texture does need to be recomputed on the atlas. Our implementations were able to support rates of 10 Hz for an atlas of resolution 256^2 using the host processor for rasterization and texture evaluation. These speeds will improve as graphics hardware performance continues to accelerate.

Team LRN

ACKNOWLEDGMENTS

Nate Carr was responsible for most of the work described in this chapter. Our work on this topic was supported in part by the Evans & Sutherland Computer Corp. Jerome Maillot was instrumental in developing the initial ideas behind this work.

Team LRN

14

A BRIEF INTRODUCTION TO FRACTALS

F. KENTON MUSGRAVE

Our world is visually complex. Achieving realism in computer graphics is largely a matter of reproducing that complexity in our synthetic images. *Fractal geometry* is our first cogent language of visual complexity; it provides a potent vocabulary for complex form, particularly the kinds of forms found in nature. Fractal geometry can map seemingly chaotic complexity into the terse, deterministic idiom of mathematics—simple equations that efficiently encapsulate lots of complexity. Furthermore, the way in which fractals encapsulate this complexity is exquisitely suited to the capabilities of the digital computer (unlike much mathematics). Computer graphics, for its part, can translate the fractal mathematical abstractions into the single form best suited to human cognition: images.

Fractals and computer graphics have grown up together, and both are relatively new disciplines. This is partly because the study of fractals is simply not possible without computer graphics: fractals are too complex to comprehend except through pictures and too tedious to create except with a computer. Thus computers have always been essential to the study of fractals. For their part, fractals have always been the source of much of the visual complexity in realistic computer graphics. Fractals are particularly salient in synthetic images of nature such as landscapes, while fractal textures are often used to add visual interest to relatively simple and boring geometric models. Fractals can even comprise abstract art in themselves, as Figures 14.1, 19.4, and 19.5 illustrate. It is safe to say that fractals have been the source of much of the beauty in synthetic images throughout the brief history of computer graphics. Fractals and computer graphics go hand in hand.

This chapter provides a brief overview of fractal geometry. It is designed to be a sort of "fractals for artists" discussion of the relevant issues. I have tried hard to be accurate in the details while avoiding mathematical technicalities, knowing how stultifying they can be. The mathematics is covered in exquisite detail in the excellent text *The Science of Fractal Images* (Peitgen and Saupe 1988) if you're interested.

Team LRN

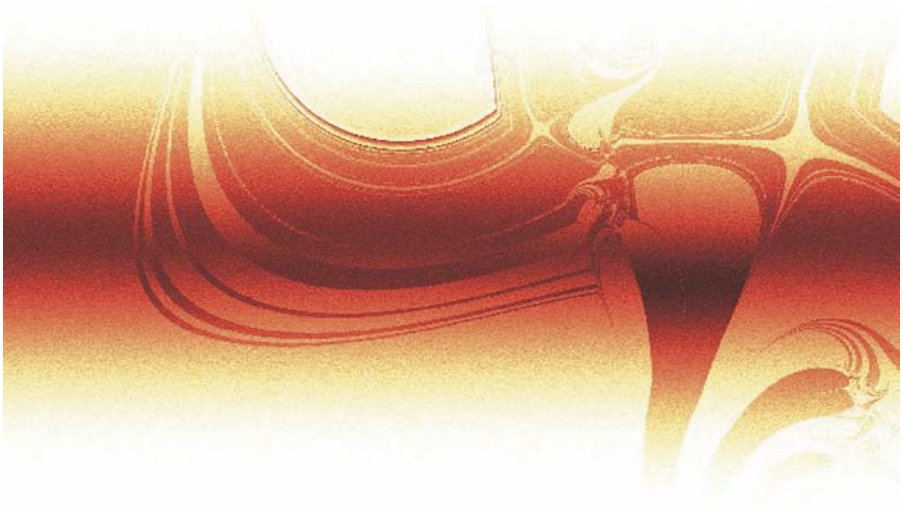


FIGURE 14.1 "Genetic Sand Painting" is a procedural texture "grown" using Karl Sims's genetic software. Copyright © 1994 F. Kenton Musgrave.

The importance of this chapter lies in the introduction of two fractal constructions and the code segments that implement them. Let me state up front that the second, pure multifractal, function may safely be ignored as simply a mathematical curiosity, but the first is *the* primary building block for all of the constructions I will present in later chapters. So, whatever your level of technical and/or mathematical competence, I urge you to read on. I hope you enjoy the discussion; I have attempted to make it provocative as well as informative.

Finally, a warning: Don't expect to get all this your first time through. Expect to have to read it maybe three times before you really get it. It took me at least that many times when I was first grappling with the weirdly counterintuitive concepts behind fractals. Don't feel slow or dense—you're not alone! I like to tell my students, "Many things are obvious—after you've thought about them for several years." So take your time. There's some major conceptual weirdness coming right up.

WHAT IS A FRACTAL?

As important as fractals are to computer graphics, they have always been widely misunderstood in our field. At times they have even become a source of heated controversy. Usually, in my experience, the problems arise mainly from incomplete

Team LRN

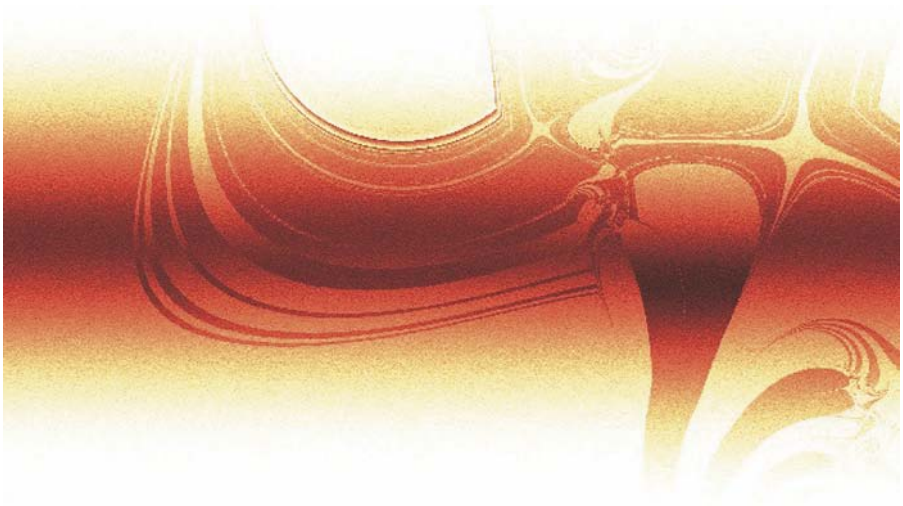


FIGURE 14.1 "Genetic Sand Painting" is a procedural texture "grown" using Karl Sims's genetic software. Copyright © 1994 F. Kenton Musgrave.

The importance of this chapter lies in the introduction of two fractal constructions and the code segments that implement them. Let me state up front that the second, pure multifractal, function may safely be ignored as simply a mathematical curiosity, but the first is *the* primary building block for all of the constructions I will present in later chapters. So, whatever your level of technical and/or mathematical competence, I urge you to read on. I hope you enjoy the discussion; I have attempted to make it provocative as well as informative.

Finally, a warning: Don't expect to get all this your first time through. Expect to have to read it maybe three times before you really get it. It took me at least that many times when I was first grappling with the weirdly counterintuitive concepts behind fractals. Don't feel slow or dense—you're not alone! I like to tell my students, "Many things are obvious—after you've thought about them for several years." So take your time. There's some major conceptual weirdness coming right up.

WHAT IS A FRACTAL?

As important as fractals are to computer graphics, they have always been widely misunderstood in our field. At times they have even become a source of heated controversy. Usually, in my experience, the problems arise mainly from incomplete

Team LRN

knowledge of exactly what fractals are and are not. They *are* a potent language of form for shapes and phenomena common in nature. They are decidedly *not* the end-all for describing all aspects of the world we inhabit, or for creating realistic synthetic images. They encompass simultaneously more and less than what many people think. I will attempt to illuminate the concept of a "fractal" in the following, and also to make a first cut at delineating both the power of fractal geometry and at least some of its limitations. Having worked with Benoit Mandelbrot, the father of fractal geometry, for six years, I may be qualified to present the topic and to address some of the misunderstandings of fractals that are common in our field.

Fractal geometry is mathematics, but it is a particularly user-friendly form of math. In practice, it can be approached entirely heuristically: no understanding of the underlying formulas is required, and the numerical parameters of the user interface may be used intuitively, their exact values being more or less arbitrary "positions of a slider," a slider that controls some aspect of visual behavior. For the purposes of this text, I will develop this heuristic approach to fractals in computer graphics. For my own purposes, I think entirely visually; the equations simply describe shapes and rules for their combination. Works for me! I am *not* a mathematician—rumors to the contrary are greatly exaggerated.

What, then, *is* a fractal? Let me define a fractal as "a geometrically complex object, the complexity of which arises through the repetition of a given form over a range of scales."[1] Note the simplicity and breadth of this definition. This simple, heuristic definition will be sufficient for our purposes here.

Probably the easiest way to think of fractals is as a new form of symmetry—dilation symmetry. Dilation symmetry is when an object is invariant under change of scale—zooming in or zooming out. This invariance may be only in gross appearance, not in the exact details. For example, clouds exhibit dilation symmetry in that a small part of a cloud looks like a larger part, but only qualitatively, not exactly. Tree branches, river networks, and lightning display this dilation symmetry, too. And all are fractal.

In my current worldview there are at least two kinds of complexity: fractal and nonfractal. Nonfractal complexity seems to be characterized by the accumulation of a variety of features through distinct and unrelated events over time—like the scuffs, holes, and stains on an old pair of shoes, for instance. Because of the independence of the events that create the features that comprise this complexity, it is hard to characterize it succinctly. Fractal complexity, on the other hand, can be very simple: just keep repeating the same thing over and over, at different scales.

1. No tricks here: "scales" simply means "sizes."

Team LRN

Our heuristic definition is sufficient, but let me explain a little further for your edification. Fractals have a peculiar property called *fractal dimension*. We are all familiar with the Euclidean integer-valued dimensions: zero dimensions corresponds to a point, one to a line, two to a plane, and three to space. Fractal dimension extends this concept to allow real-numbered values such as 2.3. Visually, values of fractal dimension between the integer values (for example, 2.3, which lies between 2 and 3 dimensions) provide a continuous "slider" for the visual complexity of the fractal. Something with a fractal dimension of 2.0 is (at least locally) planar, and as the value of the fractal dimension rises from 2.0 to 3.0, that plane becomes rougher and rougher—and more visually complex—until it densely occupies (at least locally) some volume of three-dimensional space. I warned you that this would get weird!

Again, the "whole" component of the fractal dimension, or the 2 in 2.3, indicates the underlying Euclidean dimension of the fractal, in this case a plane. The "fractional" part, for example, the .3 in 2.3, is called *the fractal increment*. As this part varies from .0 to .999 . . . , the fractal literally goes from (locally) occupying only its underlying Euclidean dimension, for instance, a plane, to densely filling some local part of the next higher dimension, such as space. It does this by becoming ever more convoluted as the value of the fractal increment increases.

Why do I keep qualifying these statements with the word "locally"? Because while a beach ball is, for example, a three-dimensional object, its surface is not: if we zoom in infinitely close, it becomes "locally" planar. Sure, it's a bit of a fine mathematical point, but understanding it and keeping it in mind is necessary to help avoid even more confusion than is inevitable in learning what fractals are. So a surface doesn't have to be a perfectly flat Euclidean plane to have a fractal dimension of 2.0, nor does it have to fill all of three-space to have a fractal dimension of 3.0. This one took me a while to get my head around, too. You're not alone.

Fractal dimension is a peculiar property, to be sure. The mathematical definition of it involves infinity; therefore, I claim, the human mind simply cannot fully comprehend it. But you quickly become numb (well, after a few years maybe) to the incomprehensibility, and furthermore, for our purposes here it is beside the point. So let us not belabor the technical details; rather, let us continue in developing an intuitive grasp of fractals.

The source of the convoluted complexity that leads to this intermediate dimensionality is, again, simply the *repetition of some underlying shape,* over a variety of different scales. I refer to this underlying shape as the *basis function*. While that function can be literally anything, for most of my constructions it is a variant of Ken Perlin's *noise* function.[2] For now, think of the noise function as providing a kind of

2. I generally prefer to use "gradient noise," as described by Darwyn Peachey in Chapter 2.

Team LRN

cottage cheese with lumps all of a particular size. We build a fractal from it simply by scaling down the lateral size of the lumps, and their height as well, and adding them back in to the original lumps. We do this several times and—presto!—we have a fractal.

To be a little more technical, we refer to the lateral size of the lumps as the *frequency* of the function (more specifically, the *spatial frequency)*. The height of the lumps we refer to as the *amplitude*. The amount by which we change the lateral size, or frequency, in each step of our iterative addition process is referred to as the *lacunarity* of the fractal. Lacunarity is a fancy Latin word for "gap." The gap, in this case, is between successive frequencies in the fractal construction. In practice, lacunarity is pretty much a nonissue, as we almost always leave it set at 2.0 (although in Chapter 6 Steve Worley describes some cases where you might want to change that value slightly). In music, doubling the frequency—which is exactly what a lacunarity value of 2.0 implies—raises a given pitch by exactly one octave. Hence we generally speak of the number of *octaves* in our fractals: this corresponds to the number of times we scaled down and added in smaller lumps to bigger lumps.

There is a well-defined relationship between the amount by which we scale size and the amount by which we scale the height, or frequency versus amplitude. This relationship is what determines the fractal dimension of our result. (Again, I decline to get any more technical and refer the interested reader to *The Science of Fractal Images* for details.) The particular kind of fractal we're building is called *fractional Brownian motion,* or fBm for short. fBm is characterized by its *power spectrum,* which charts exactly how amplitude relates to frequency. Oops! Pardon me—I'll knock off the math.

Allow me now to point out two of the most common misconceptions about fractals in our field of computer graphics. The first is that all fractals are variants of fBm. Not so! Go back and look at our definition of fractals: it subsumes a far larger class of objects than our scaled-and-added-up cottage cheese lumps. Many things are fractal; more interestingly, perhaps, many things are "only sort of" fractal. Fractalness is a property that is, believe it or not, best left loosely defined. Think of it as a quality like color: it is hard to define "blue" precisely, and any really precise definition of "blue" is likely to be overly restrictive in certain circumstances. The same goes for fractals.

The second common misconception about fractals is this: that an object must have infinite detail in order to qualify as fractal. Also not so! Benoit and I agree that to talk about self-similarity over a range of less than three scales is rather vacuous. So we may heuristically constrain fractals to be "objects that display self-similarity at a minimum of three separate scales." This magic number three may not even provide such a great definition of "fractal"—later I will describe a nice model of water

Team LRN

that can use only two octaves of noise. Is it fractal, or is it not? You could make a convincing argument either way. The main point is this: as long as something displays self-similarity over some, albeit perhaps small, range of scale, it may qualify as "fractal." Note that *all* fractals in nature exhibit their fractal behavior over a limited range of scale—even the large-scale cosmological structure of the universe. The distribution of galaxies and clusters of galaxies is quite fractal, but there *is* a finite and observable largest scale of features in this universe (knowledge more recent than the first edition of this book!). Another example: Seen from a distance in space, Earth is smoother than a glass marble, yet on smaller scales it has many mountain ranges that are quite fractal.

We refer to the size above which self-similarity ceases to manifest itself as the *upper crossover scale*. Similarly, there is a smaller size below which self-similarity no longer is manifest; this is the *lower crossover scale*. All fractals in nature, then, are what we call *band-limited*—they are fractal only over some limited range of scales. Mandelbrot makes this striking observation: the Himalayas and the runway at JFK have approximately the same fractal dimension (i.e., roughness)—they differ only in their crossover scales! (Didn't I warn you this would get weird?)

Finally, I'd like to note that all fractals for computer graphics must also be band-limited, for two reasons: First, spatial frequencies that are higher than half our pixel frequency (the screen width divided by resolution) may violate the Nyquist sampling limit and cause aliasing (a highly technical point; feel free to ignore it here and when it comes up again later, but hugely important to our ultimate goal of building the realistic fractal planets that will become cyberspace). Second, we generally wish for our computations to terminate, so it's poor programming practice to put in the kind of infinite loop that would be required to construct non-band-limited fractals.

WHAT ARE FRACTALS GOOD FOR?

Again, the world we inhabit is visually complex. When synthesizing worlds of our own, *complexity* equals *work*. This work can be on the part of the programmer/artist or the computer; in fact it will always be some of each. But there is a balance to be struck, and personally I prefer to have the computer do most of the work. Usually, it seems to have nothing better to do, while I generally can find other mischief to make. One of the defining characteristics of procedural modeling, in general, is that it tends to shift the burden of work from the programmer/artist to the computer. Complexity is vital to realism in synthetic images. We still have a long way to come in this area: I claim that you'd be hard put to find any scene in your everyday environment, other than a clear blue sky, that's as visually simple as the best, most detailed synthetic image of the same thing (if for no other reason than that reality is higher

Team LRN

resolution). How, then, can we close the gap? To date, fractals are our best tool. While not all complexity in nature comprises the repetition of form over different scales, much of it is. It is not so much true that nature is fractal as that fractals are natural. Fractal models can be used to construct scenes with good realism and a high degree of visual complexity, but they effectively address only a limited range of phenomena. People often ask me if I can model dogs and people with fractals and the answer is "no." Dogs and people simply aren't self-similar over a range of scales. Fractals, then, allow us to model certain things well: mountains, clouds, water, even planets. Non-self-similar complexity such as hair and grass require other methods. Things not visually complex, obviously, do not require fractal geometry for their reproduction.

All my time with fractals has led me to view them as "the simplest conceivable form of complexity." (If you can think of a simpler way to give rise to complexity than to simply repeat the same thing over and over at a variety of scales, I'd love to hear about it!) This simplicity is a very good thing, since computers are simpletons, and simpler programs are better programs, too: they are quicker to write, (usually) easier to understand, easier to trust, and take up less memory.[3]

Now let me answer the question "What are fractals good for?" Many natural phenomena are fractal. Mountains are perhaps the best-known example: a smaller part of a mountain looks just as mountainlike as a larger part. They're not exactly the same; this is the distinction between *statistical self-similarity,* where only the statistics of a random geometry are similar at different scales, and *exact self-similarity,* where the smaller components are exactly the same as the larger ones, as with the equilateral triangles forming the famous von Koch snowflake fractal. Trees, river systems, lightning, vascular systems in living things—all have the character of statistical self-similarity: smaller branches tend to resemble larger branches quite closely. Another example, on which we capitalize quite heavily in this book, is turbulent flow: the hierarchy of eddies in turbulence was known to be self-similar long before Mandelbrot elucidated the concept of "fractal." There's even a little poem about it:

> Bigger swirls have smaller swirls,
> That feed on their velocity,
> And smaller swirls have smaller swirls,
> And so on, to viscosity
> —Lewis F. Richardson, 1922

3. This bias in favor of simplicity is canonized in Occam's Razor: "The simpler model is the preferred model." This principle has guided much of science and engineering through the centuries. More on this later.

Team LRN

The essential fractal character of turbulence allows us to use fractal models to represent clouds, steam, global atmospheric circulation systems on planets, even soft-sediment deformation in sedimentary rock. Again, the pictures attest to the success of these models, as well as the limits to that success.[4]

FRACTALS AND PROCEDURALISM

How are fractals and proceduralism related? Very closely indeed. When we describe how to build fBm, it will be with an iterative, constructive procedure. The simplicity of fractals resonates well with computers, too, as they are simple-minded devices. The sort of simple, tedious, repetitive operations required to build a fractal are exactly the kind of thing computers do best. As we will see, fractal constructions can provide potentially unlimited visual complexity that issues from a relatively small amount of code.

Alvy Ray Smith called this complexity-from-simplicity *amplification* (Smith 1984)—a small input provides a wealth of output. We engineer it in the classic proceduralist method: by shifting the burden of work from the human designer to the computer. Mandelbrot himself points out that fractals have been evident to mathematicians for some time, but not until the advent of computer graphics did they have the tools necessary to investigate them. This is almost certainly why fractals were only recently "discovered"—perhaps more like "fleshed out"—and why it was a researcher (Mandelbrot) at IBM (a computer company) who did so. Fractals and computers have always been inextricably linked. The fruit of this symbiosis is illustrated in the realistic synthetic imagery issuing from fractal models and by the observation that most other complexity in computer graphics derives either from captured data from the real world (cheating!) or from the hard labor of constructing detailed models by hand (which is antiproceduralist).

PROCEDURAL fBm

But enough lecturing already. Let's see exactly how to build the archetypal fractal procedural texture: fBm (also known as "plasma" in some software applications). It's pretty simple:

4. For instance, as our poem points out, turbulence is actually composed of a hierarchy of *vortices,* not simply lumps like the *noise* function provides. To my knowledge, no one has yet developed a procedural vortex turbulence model that competes well with physical simulations using the Navier-Stokes equations that yield nice fractal vortices, but at a high computational cost.

Team LRN

```
/*
 * Procedural fBm evaluated at "point".
 *
 * Parameters:
 * "H" is the fractal increment parameter
 * "lacunarity" is the gap between successive frequencies
 * "octaves" is the number of frequencies in the fBm
 */
double fBm( Vector point, double H, double lacunarity, double octaves )
{
     double         value, remainder, Noise();
     int            i;

     value = 0.0;

     /* inner loop of fractal construction */
     for (i=0; i<octaves; i++) {
     value += Noise( point ) * pow( lacunarity, -H*i );
          point *= lacunarity;
     }

     remainder = octaves - (int)octaves;
     if ( remainder ) /* add in "octaves" remainder */
          /* 'i' and spatial freq. are preset in loop above */
          value += remainder * Noise3( point ) * pow( lacunarity, -H*i );
     return value;
}
```

Note the simplicity of this routine: the fractal itself is constructed in the two-line inner loop; the rest of it is concerned with the picayune point of dealing with the remainder of `octaves`. (This will be important when building fractals with continuous level of detail, or LOD, in later chapters.[5]) Again, this routine is a generalization of Perlin's original "chaos" function. I've added new parameters to control lacunarity (which in most cases can simply be fixed at 2.0, as Perlin originally did), the fractal increment parameter *H*, and the number of octaves in the construction.

---

5. For a properly band-limited fBm with pixel-level detail, imaged from a distance of 1.0, the correct number of octaves to use is

$$\texttt{octaves} = \log_2(\texttt{screen\_resolution}) - 2$$

or a value of about 6 to 10. The −2 term in this expression has to do with the facts that the Nyquist limit is 1/2 of the screen resolution and that the highest frequency in the Perlin *noise* function is ~1/2 of the lattice spacing. Then we have 1/2 * 1/2 = 1/4 and $\log_2(1/4) = -2$. Reducing the number of octaves can speed up rendering time in exchange for less detail.

Team LRN

To accommodate LOD in advanced applications, this function is designed to accommodate real-valued octaves. The fractional part of the value of parameter `octaves` is added in, linearly, after the main loop. We'll use this to adaptively band-limit textures where we want to link the number of octaves to distance to avoid exceeding the Nyquist limit and subsequent aliasing. (We will utilize this in the QAEB algorithm in Chapter 17.) This little trick with the octaves remainder avoids discontinuities in the texture that would be introduced were the number of octaves to abruptly change at some threshold.

If you implement something like adaptive band-limiting, you'll want to store the spectral exponents in a preinitialized array, to avoid repeated calls to `pow()` in the inner loop. This exponent array would store the amplitude-scaling values for the various frequencies in the construction. These weights, a simple function of H and the lacunarity, determine the fractal dimension of the function. (Again, see *The Science of Fractal Images* for details on how and why.)

The parameter H is equal to one minus the fractal increment. It corresponds to the H described by Voss in *The Science of Fractal Images*. When $H = 1$, the fBm is relatively smooth; as H goes to 0, the function approaches white noise. Figure 14.2 shows traces of the function for various values of H.

The underlying Euclidean dimension of the function is given by the dimensionality of the vector-valued argument `point`.[6]

MULTIFRACTAL FUNCTIONS

The fBm described above is, at least as closely as we can make it (see Chapter 6 for more on this), statistically *homogeneous* and *isotropic*. Homogeneous means "the same everywhere," and isotropic means "the same in all directions" (note that the two do not mean the same thing). Fractal phenomena in nature are rarely so simple and well behaved. For instance, synthetic fractal mountains constructed with a single, uniform fractal dimension everywhere—as are most fractal mountains in computer graphics—have the same roughness everywhere. Real mountains are never like that: they typically rise out of plains and have rolling foothills at their feet. For

6. A definition of *vector:* an ordered set of numbers, usually three numbers for our purposes, which defines an arrow from the origin (the origin being [0,0,0] in three dimensions) to the point in space that the set of numbers specifies. The dimensionality of the space is the same as the number of elements in the vector. Thus the vector [1,−1,1] defines an arrow in three dimensions that points out from the origin at 45 degrees to each of the x-, y-, and z-axes. Its length is $\sqrt{x^2 + y^2 + z^2} = \sqrt{1^2 + (-1)^2 + 1^2} = \sqrt{3}$.

Team LRN

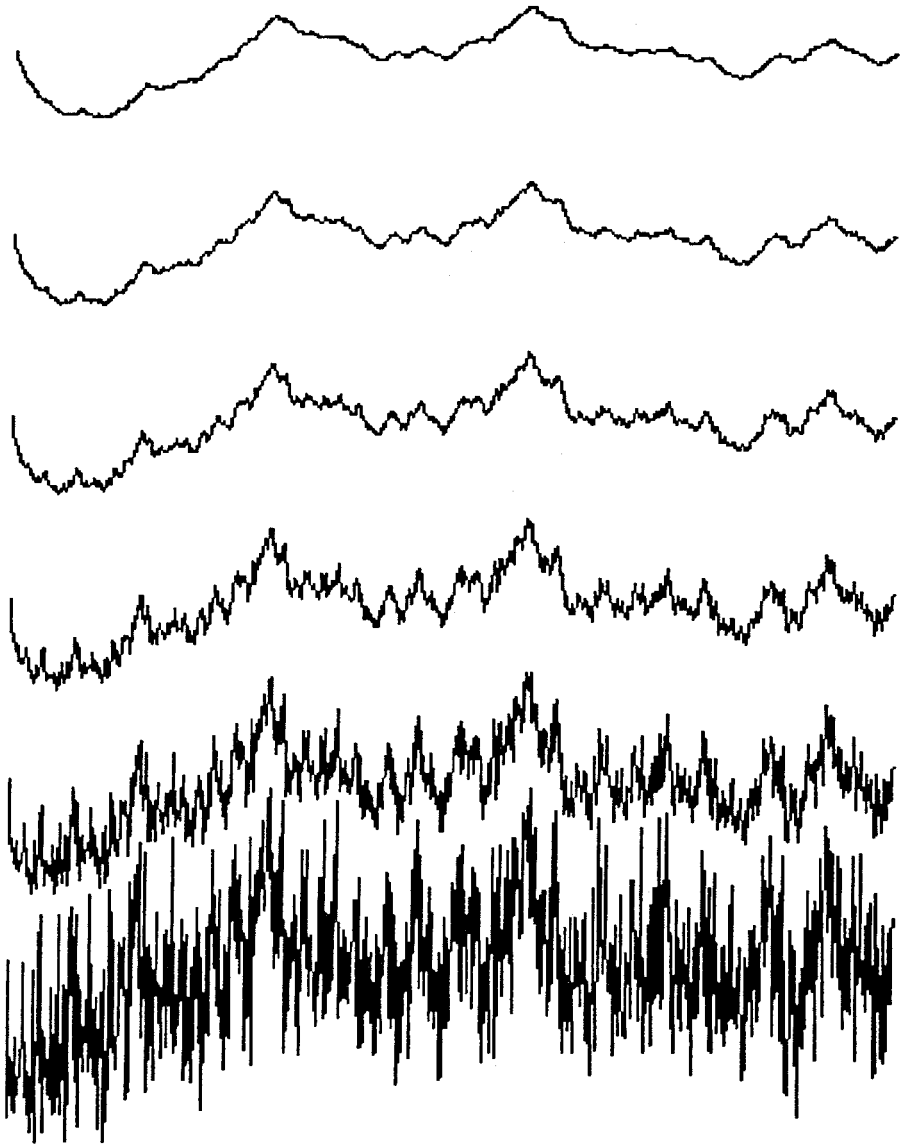


FIGURE 14.2 Traces of fBm for H varying from 1.0 to 0.0 in increments of 0.2.

Team LRN

this and other concerns of realism, we desire some more interesting, heterogeneous fractal functions.

Enter *multifractals*. Multifractals may be heuristically defined as "fractals that require a multiplicity of measures, such as fractal dimension, to characterize them." Another heuristic definition is "heterogeneous fractals, the heterogeneity of which is invariant with scale."[7] They are most easily thought of as fractals whose dimension varies with location; the key point is that they are heterogeneous—not the same everywhere. Later I will demonstrate some terrain models with plains, rolling foothills, and jagged alpine mountains, all of which issue from a single function that is only a little more complicated than the basic fBm described earlier.

One mathematical definition of multifractals ties them to *multiplicative cascades* (Evertsz and Mandelbrot 1992). The earlier fBm function is built using an *additive cascade*. The formal difference between an additive and a multiplicative cascade is simply that the addition of higher frequencies in the inner loop is replaced by a multiplication. Here is a multiplicative-cascade multifractal variation on fBm:

```
/*
 * Procedural multifractal evaluated at "point."
 *
 * Parameters:
 * "H" determines the highest fractal dimension
 * "lacunarity" is gap between successive frequencies
 * "octaves" is the number of frequencies in the fBm
 * "offset" is the zero offset, which determines multifractality
 */
double
multifractal( Vector point, double H, double lacunarity,
              int octaves, double offset )
{
    double    value, Noise();

    value = 1.0;

    for (int i=0; i<octaves; i++) {
         value *= (Noise( point ) + offset) * pow( lacunarity, -H*i );
         point *= lacunarity;
    }
    return value;
}
```

7. The heterogeneity may be, for instance, that peaks on a terrain are rougher than valleys. This can be true at all scales; then we have a multifractal, by this definition.

Team LRN

Note the addition of one more argument, `offset`, to the function, over the function for ordinary *(monofractal)* fBm; other than this and the multiplication in the inner loop, this function is nearly identical to `fBm( )`. The `offset` controls the multifractality, if you will, of the function. When `offset` equals zero, the function is heterogeneous in the extreme, and as its value increases the function goes from multi- to monofractal, and then approaches a flat plane as `offset` gets large (i.e., around 100 or so). An `offset` value of ~0.8 yields a very nice, heterogeneous terrain model, as seen in Figure 14.3. One thing you must know about this function: its range varies widely with changes to the value of `offset`—by many orders of magnitude. In fact, as the number of octaves goes up, it will either converge to zero or diverge to infinity. It's just plain unstable. (Hence the parameter `octaves` is not real-valued, as this function would behave badly in LOD schemes.) If you ever use this function, you will need to take measures to rescale its output. I have accomplished this by evaluating the function over some finite patch (e.g., a 3 × 3 area sampled at 100 × 100 resolution) and rescaling the function's output by one over the maximum value in the patch.

This function has the abstract advantage of following at least one mathematical definition of "multifractal" closely. This may be desirable for mathematical research into multifractals, but it is really pretty much a red herring for those more interested in applications. Therefore, I suggest that you just ignore this construction and concentrate on the less mathematically well-defined, but better behaved, multifractal

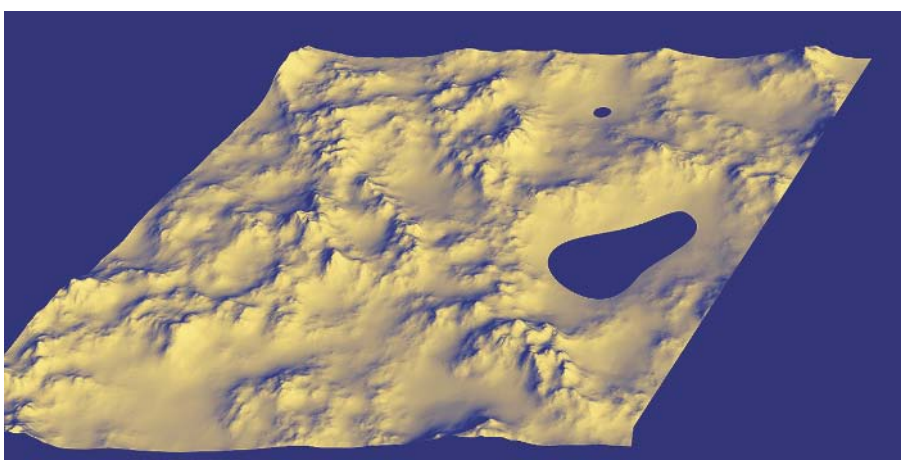


FIGURE 14.3 A multifractal terrain patch, with `offset` parameter set to 0.8. Copyright © 1994 F. Kenton Musgrave.

Team LRN

functions described in Chapter 16. Those functions are additive/multiplicative hybrids that we don't quite know how to characterize mathematically yet, but that have proven their usefulness in applications.

These multifractal models were developed for terrain models, and I have most often interpreted them as height fields for that purpose. But in fact they are really just more examples of procedural textures, and they can be used as such. I encourage you to experiment with them and invent your own new applications—it's a lot of fun! (For example, see Figure 15.17.) Since they were conceived as terrain models, I will explain them in that context, in Chapter 16. But keep in mind that those models can also be used as textures when applications arise.

FRACTALS AND ONTOGENETIC MODELING

Fractal models are sometimes assailed on the grounds that they lack a physical basis in reality. For example, there is no known causal basis for the resemblance between fBm and real mountains. Is this a problem? For the reductionist-minded it is, for such a model violates the reductionist principle that true validity can only be established by showing that the model issues from first principles of physical law. This is a narrow view that Mandelbrot and I, among others, would dispute. To wit, Nobel Prize–winning physicist Richard Feynman once said: "A very great deal more truth can become known than can be proven" (Brown and Rigden 1993). In practice, both physical and nonphysical modeling have their places in computer graphics. To provoke and focus this argument, I coined the term *ontogenetic modeling*. From *Webster's Collegiate Dictionary,* 10th edition:

> *ontogenetic:* . . . 2: based on visible morphological characters.

I coined the term deliberately to contrast with Al Barr's "teleological" modeling (Barr 1991). Again, from *Webster's:*

> *teleology:* 1 a: the study of evidences of design in nature b: a doctrine (as in vitalism) that ends are immanent in nature c: a doctrine explaining phenomena by final causes 2: the fact or character attributed to nature or natural processes of being directed toward an end or shaped by a purpose 3: the use of design or purposes as an explanation of natural phenomena.

Team LRN

The underlying idea of ontogenetic modeling is that, in the field of image synthesis, it is a legitimate engineering strategy to construct models based on subjective morphological (or other) *semblance;* this as opposed to, for instance, pursuing precise (e.g., mathematical) veracity, as is a goal in scientific models, or constructing them such that they and their behavior issue from first scientific principles, in the reductionist tradition.

My point is that we computer graphics professionals are engineers, not scientists. The goal of engineering is to construct devices that do something desirable. The goal of science is to devise internally consistent models that reflect, and are consistent with, the behavior of systems in nature (which are entirely unrelated to such models) and to make verifiable predictions of the behavior of nature based on the behavior of the model. Science informs engineering. But engineering is an ends-driven discipline: if the device accomplishes what it is intended to accomplish, it works and is therefore good. The means by which it accomplishes its end are of secondary importance.

Elegance[8] in the model is nice in engineering, while it is necessary in science. Occam's Razor (i.e., the metaphysical prejudice that "the simpler solution is the preferred solution") is equally applicable in both science and engineering. Engineers call it the KISS principle: "Keep it simple, stupid."[9]

In engineering—specifically, in computer graphics—Occam's Razor often recommends ontogenetic models. Science is going for Truth and Beauty; we're after beautiful pictures that look like things familiar. The degree of preoccupation with accuracy and logical consistency that marks good science is not admissible in our engineering discipline: we have a job to get done (making pictures); getting that job done efficiently takes precedence. Scientific models, or "physical" models in computer graphics parlance, do not generally map well into efficient algorithms for image synthesis. Given the aims and methods of science, this is not surprising. Algorithmic computability and/or efficiency are not considerations in constructing scientific models. And concerns for efficiency invoke a new, different set of constraints that may be orthogonal to the considerations that shaped any given scientific model. For instance, no scientist would hesitate to use an integral with no closed-form solution in a model—the lack of a mechanism to obtain an exact evaluation is

8. Again from *Webster's:* "elegance: scientific precision, neatness, and simplicity."

9. My mentor in landscape photography, Steve Crouch, put it another way: referring to composing an image in the viewfinder, he said "See what you can get out of the picture," not what you can get into it.

Team LRN

orthogonal to, and in no way compromises the validity of, the model. In the engineering discipline of image synthesis, we *must* be able to complete our computations, faster is better, and if it looks good enough, it *is* good enough.

As the late Alain Fournier put it: When you use a physical model for image synthesis, you often waste time computing the answers to questions about which you care not. That is, they do not contribute to the image. Excessive accuracy in illumination calculations is an example, given the fact that we quantize our illumination values to at most 256 levels in most standard output formats.

Given the serious drawbacks and complications of physically based models of natural phenomena, I claim that the ontogenetic approach remains, for the foreseeable future, a viable alternative approach to engineering the synthesis of realistic images. Ontogenetic models tend to be—indeed *ought* to be—simpler and more efficient, both in programming and execution time, than corresponding physical models.

[Editorial note, circa 2002: I got on my soapbox to deliver that rant about 15 years ago, when I was a graduate student and this was a hot topic. By now it seems pretty dated and my passion, well, quaint. But the basic points remain valid and well taken, I think. —FKM]

CONCLUSION

I hope I've helped you establish an intuitive understanding of fractals in this chapter. The level of understanding presented here is all that is really required to pursue applications. You don't need to be a mathematician to create and use fractals; in fact, my experience indicates that an artistic eye is far more important than a quantitative facility for creating striking fractal models for synthetic imagery. Keep in mind that fractals are the simplest and easiest—for the human operator, at least—way to generate visual complexity, whether it be geometric detail, as in landscapes, or visual detail, as with procedural textures. Fractals represent a first step toward procedurally elegant descriptions of complexity in the natural world.

To work effectively with fractals, you need to be familiar with the heuristic definition of *fractal dimension*—merely that it corresponds to roughness or wiggliness. You need to be aware of the idea of *octaves*—the number of scales at which you're adding in smaller details. Also, you may occasionally find it helpful to be familiar with the concept of *lacunarity*—the change in scale between successive levels of detail—although you probably will never have need to use this knowledge.

Team LRN

Finally, it is helpful to understand that most fractal constructions we use in computer graphics are *monofractal*—that is, they are homogeneous, and for that reason, may become a bit monotonous and boring. *Multifractals* can provide a second step toward capturing more of the true complexity abundantly manifest in nature. Turbulence, for instance, is a decidedly multifractal phenomenon.

With these elements of understanding in place, let us proceed to applications and see how fractals may be used in procedural models.

Team LRN

15

Team LRN

FRACTAL SOLID TEXTURES: SOME EXAMPLES

F. KENTON MUSGRAVE

This chapter will describe some fractal procedural textures serving as models of natural phenomena. They are divided into the four elements of the ancients: air, fire, water, and earth. The presentation format of the code segments, for the most part, has been switched from the C++ programming language to the RenderMan (Upstill 1990) shading language.[1] The fractal functions described in the previous chapter are to be used as primitive building blocks for the textures we'll develop here, and as such, they should be implemented in the most efficient manner possible (e.g., in compiled code). If you like any of the textures we develop here enough to make them part of a standard texture library, you might want to translate them to compiled code, for efficiency. But, in general, it is quicker and easier to develop texture functions using higher-level tools such as the function graph editors in MojoWorld and Dark Tree or the RenderMan shading language. Take the texture constructions described here as starting points and modify them. Do lots of experiments and come up with your own unique textures—after all, such experimentation is how these textures came into being! Nothing here is written in stone; you can have endless fun devising variations.

The textures described in this chapter were certainly not designed as a whole, a priori, then implemented. Rather, they are generally the result of "hacking"—hours and hours of making modifications and extensions, evaluating the texture to see how it looks, making more changes, and so on. It may seem to you that this iterative loop is more artistic than scientific, and I would agree that it is, but it does share with the scientific method what Gregory Nielson calls "the basic loop of scientific discovery" (Nielson 1991): you posit a formal model (nothing, after all, is more formal than the

1. The translations from C to the RenderMan shading language are provided courtesy of Larry Gritz of ExLuna, the author of the shareware RenderMan package Blue Moon Rendering Tools (BMRT). Larry kindly updated the shaders for the second edition of this book. They have been tested and verified on both Pixar's PhotoRealistic RenderMan and BMRT.

Team LRN

logic of a computer program), you make observations of the behavior of the model and the system being modeled, you make modifications to improve the model, then more observations, and so on, in an iterative loop. Perhaps the main difference between science and computer graphics is, as Pat Hanrahan has pointed out, in the time required per iteration: that time is certainly much shorter when designing procedural textures than when pursuing the physical sciences. The point is, no scientific model was born perfect, and neither is any complex procedural texture. Particularly in ontogenetic modeling, again, we are not concerned with Truth but rather with visual Beauty. We are more interested in semblance than veracity. When you are more interested in the quality of the final image than in the methodology of its production, you are more an engineer than a scientist. In such an endeavor, whatever gets us the result in a reasonable amount of time, both human and computer, is a viable strategy. So we'll do whatever works.

CLOUDS

Clouds remain one of the most significant challenges in the area of modeling natural phenomena for computer graphics. While some very nice images of clouds have been rendered by Geoffrey Gardner, Richard Voss, David Ebert, Matthew Fairclough, Sang Yoon Lee, and myself, in general the modeling and rendering of clouds remains an open problem. I can't claim to have advanced the state of the art in cloud modeling much, but I have devised a few two-dimensional models[2] that are at least significant *aesthetically.* I'll describe them below, but first I'll describe the most common, quick, and easy cloud texture.

Puffy Clouds

One of the simplest and most often used fractal textures is a simple representation of thin, wispy clouds in a blue sky. All this consists of is a thresholded fBm. If our fBm function outputs values in the range [−1, 1], we might specify that any value less than zero represents blue sky. This will accomplish the effect:

```
surface
puffyclouds(float Ka = 0, Kd = 0;
   float txtscale = 1;
   color skycolor = color(.15, .15, .6);
   color cloudcolor = color(1, 1, 1);
```

2. This may be a little confusing: I call these models 2D because, while they are implemented as 3D solid textures, they are designed to be evaluated on 2D surfaces. In Chapter 17, I'll show how to evaluate them as volumetric hypertextures via the QAEB algorithm.

Team LRN

```
    float octaves = 8, H = 0.5, lacunarity = 2;
    float threshold = 0; )
{
    float value;
    color Ct; /* Color of the surface */
    point PP; /* Surface point in shader space */

    PP = txtscale * transform("shader", P);
    /* Use fractional Brownian motion to compute a value for this point */
    value = fBm(PP, filterwidthp(PP), octaves, lacunarity, H);
    Ct = mix(skycolor, cloudcolor, smoothstep(threshold, 1, value));

    /* Shade like matte, but use color Ct */
    Ci = Ct; /* This makes the color disregard the lighting */

    /* Uncomment the next line if you want the surface to be lit */
    /* Ci = Ct * (Ka * ambient() + Kd * diffuse(faceforward(N,I))); */
}
```

This can make a good background texture for images where the view is looking up at the sky, as in Jules Bloomenthal's image of the mighty maple, shown in Figure 15.1 (Bloomenthal 1985).

FIGURE 15.1 Jules Bloomenthal's image of "the mighty maple" shows a typical use of the puffy clouds texture.

Team LRN

A Variety of fBm

As noted in the previous chapter, our fractals can be constructed from literally *any* basis function; the basis function we most often choose is the Perlin *noise* function. For theoretical reasons outlined in the previous chapter, with our usual lacunarity of 2.0, we use at most about 8 to 10 octaves of the basis function in our constructions. It turns out that the character of the basis function shows through clearly in such a small spectral summation. So, if we use a different basis function, we get fBm with a different character, with a different aesthetic "feel" to it. (That's why there are over 200 different basis functions available in MojoWorld, with practically infinite variations on them possible.) Take, for example, fractal mountains constructed by the popular polygon subdivision schemes. The basis function implicit in the linear interpolation between the sample points is a sawtooth wave. Thus the resulting mountains are always quite triangular and jagged. When it comes to playing with different basis functions, the possibilities are endless. Wavelets (Ruskai 1992) offer an exciting prospect for basis function manipulation; Lewis's "sparse convolution" (Lewis 1989), also called "fractal sum of pulses" in Lovejoy and Mandelbrot (1985), is a theoretically desirable approach that accommodates the use of any finite basis function. Unfortunately, it is slow in practice. Try it in MojoWorld and see for yourself.

Let me now describe one trick I sometimes play with the Perlin *noise* function, to get a variation of it with significantly altered aesthetic "feel." I call it, for lack of a better name, `DistNoise( )`, for *distorted noise*. It employs one of the oldest tricks in procedural textures: domain distortion. As it is a fundamental building block type function, I present it in C++.

```
double
DistNoise( Vector point, double distortion )
{
        Vector offset, VecNoise();
        double Noise();

        offset = point + 0.5; /* misregister domain for distortion */
        offset = VecNoise( offset ); /* get a random vector */
        offset *= distortion; /* scale the distortion */
        /* "point" is the domain; distort domain by adding "offset" */
          point += offset;

        return Noise( point );  /* distorted domain noise */
}
```

The function `VecNoise()` is a vector-valued noise function that takes a 3D vector as an argument and returns a 3D vector. This returned vector corresponds

Team LRN

to three separate noise values for each of its *x*, *y*, and *z* components. It can be constructed by simply evaluating a noise function at three different points in space: the one passed in as the argument and two displaced copies of it, as in the previous misregistration.

```
Vector
VecNoise( Vector point )
{
        Vector result;
        double Noise();

        result.x = Noise( point );
        result.y = Noise( point + 3.33 );
        result.z = Noise( point + 7.77 );

        return result;
}
```

Note that this is not the same thing as Perlin's `DNoise()`, which is specified to return the three partial derivatives of `Noise()`. The latter will be C-2 continuous, while our function is C-3 continuous.[3]

We displace the point passed to `VecNoise()` by 0.5 in each of *x*, *y*, and *z*, to deliberately misregister the underlying integer lattices upon which `VecNoise()` and `Noise()` are evaluated as, if you're using gradient noise, both functions have value zero at these points. Next we evaluate `VecNoise()` at the displaced point and scale the returned vector by the distortion parameter. Then we add the resultant vector to the input point; this has the net effect of distorting the input domain given to the `Noise()` function. The output is therefore also distorted.

Figure 15.2 illustrates the difference between undistorted `Noise()` and `DistNoise()`, with the distortion parameter set to 1.0. It also illustrates the difference between fBms constructed using `Noise()` and `DistNoise()`, respectively, as the basis function. That difference is subtle, but significant. To my artist's eye, the latter fBm has a sort of wispy character that looks more like certain cirrus clouds than does the "vanilla" fBm, which seems a little bland in comparison. Note, however, that `DistNoise()` is about four times as expensive to evaluate as `Noise()`, so you pay for this subtle difference. Cost aside, I have used the modified fBm to good effect in clouds, as seen in Figures 20.4 and 20.9. In the following I will refer to such fBm by the function name `DistfBm()`.

3. This business of C-2 and C-3 continuity is a mathematical point about the number of continuous derivatives a function has. If you don't know what a derivative is, just ignore this rigmarole.

Team LRN

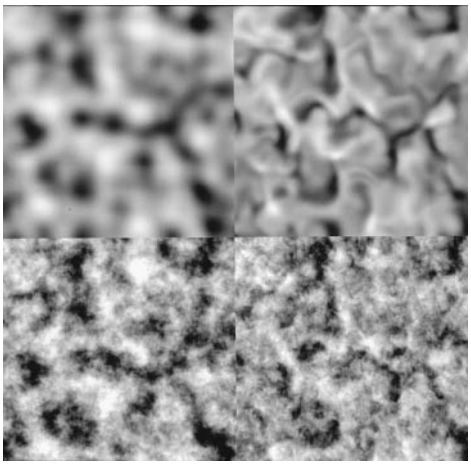


FIGURE 15.2 The upper-left square shows `Noise()`; the upper right shows `DistNoise()`. Below appears fBm constructed from each; note the subtle difference.

Note that we can write `DistNoise()`more tersely:

```
/*
 * Terse version of DistNoise( )
 */
double DistNoise( Vector point, double distortion )
{
       return Noise( point + distortion * VecNoise( point + 0.5 ) );
}
```

The RenderMan version looks very similar:

```
/*
 * RenderMan version of DistNoise( )
 *
 * Since the noise( ) function in RenderMan shading language has range
 * [0,1], we define a signed noise to be the noise function that Perlin
 * typically uses.
 */
#define snoise(P) (2*noise(P) - 1)

float DistNoise( point Pt, float distortion )
{
   point offset = snoise( Pt + point(0.5, 0.5, 0.5) );
   return snoise( Pt + distortion * offset );
}
```

```
/* Alternatively, it can be defined as a macro: */

#define DistNoise(Pt,distortion) \
   (snoise( Pt + distortion*snoise( Pt+point(0.5,0.5,0.5) ) )
```

Distortion for Cirrus Clouds and Global Circulation

Note that the previous construction is an example of *functional composition*, wherein functions become the arguments to other functions. Such nesting is a powerful procedural technique, as Karl Sims (1991) showed in his genetic LISP program, which creates procedural textures and which was used to create Figure 14.1; we'll investigate this paradigm in detail in Chapter 19. The idea of composition of noise functions to provide distortion has proved useful in another aspect of modeling clouds: emulating the streaming of clouds that are stretched by winds. We can get the kind of distinctive cirrus clouds seen in Figure 20.9 from the following specification:

```
/* Use signed, zero-mean Perlin noise */
#define snoise(x) ((2*noise(x))-1)

/* RenderMan vector-valued Perlin noise */
#define vsnoise(p) (2 * (vector noise(p)) - 1)

/* If we know the filter size, we can crudely antialias snoise by
 * fading to its average value at the Nyquist limit.
 */
#define filteredsnoise(p,width) (snoise(p) * (1 - smoothstep(0.2,0.6,width)))
#define filteredvsnoise(p,width) (vsnoise(p) * (1-smoothstep(0.2,0.6,width)))

surface
planetclouds(float Ka = 0.5, Kd = 0.75;
             float distortionscale = 1;
             float H = 0.7;
             float lacunarity = 2;
             float octaves = 9;
             float offset = 0;)
{
    vector Pdistortion;   /* The "distortion" vector */
    point PP;             /* Point after distortion */
    float result;         /* Fractal sum is stored here */
    float filtwidth;

    /* Transform to texture coordinates */ PP =
    transform("shader", P);
    filtwidth = filterwidthp(PP);
```

Team LRN

```
    /* Get "distortion" vector */
    Pdistortion = distortionscale * filteredvsnoise(PP, filtwidth);
    PP = PP + Pdistortion;
    filtwidth = filterwidthp(PP);

    /* Compute fBm */
    result = fBm(PP, filtwidth, octaves, lacunarity, H);

    /* Adjust zero crossing (where the clouds disappear) */
    result = clamp(result+offset, 0, 1);

    /* Scale density */
    result /= (1 + offset);

    /* Modulate surface opacity by the cloud value */
    Oi = result * Os;
    /* Shade like matte, but with color scaled by cloud opacity */
    Ci = Oi * (Ka * ambient() + Kd * diffuse(faceforward(normalize(N),I)));
}
```

Note that this is the same domain distortion idea that was used in `DistNoise()`, except that the distortion has greater magnitude and is at a large scale relative to the fBm.

This large-scale distortion was originally designed to provide a first approximation to the global circulation patterns in Earth's clouds as seen from space. The preceding code produced the clouds seen in Figures 20.4 and 20.9. While not a bad first approximation, it saliently lacks the eddies and swirls generated by vortices in turbulent flow.

It occurred to me that we could use an fBm-valued distortion to better emulate turbulence:

```
#define snoise(p) (2 * (float noise(p)) - 1)
#define vsnoise(p) (2 * (vector noise(p)) - 1)
#define filteredsnoise(p,width) \
   (snoise(p) * (1 - smoothstep(0.2,0.6,width)))
#define filteredvsnoise(p,width) \
   (vsnoise(p) * (1-smoothstep(0.2,0.6,width)))

/* A vector-valued antialiased fBm. */
vector
vfBm(point p;
     float filtwidth;
     uniform float maxoctaves, lacunarity, gain)
{
   uniform float i;
   uniform float amp = 1;
   varying point pp = p;
```

```
    varying vector sum = 0;
    varying float fw = filtwidth;

    for (i = 0; i < maxoctaves && fw < 1.0; i += 1) {
        sum += amp * filteredvsnoise(pp, fw);
        amp *= gain;
        pp *= lacunarity;
        fw *= lacunarity;
    }
    return sum;
}

surface
planetclouds(float Ka = 0.5, Kd = 0.75;
             float distortionscale = 1;
             float H = 0.7;
             float lacunarity = 2;
             float octaves = 9;
             float offset = 0;)
{
    vector Pdistortion; /* The "distortion" vector */
    point PP;           /* Point after distortion */
    float result;       /* Fractal sum is stored here */
    float filtwidth;

    /* Transform to texture coordinates */
    PP = transform("shader", P);
    filtwidth = filterwidthp(PP);

    /* Get "distortion" vector */
    Pdistortion = distortionscale * VfBm(PP, filtwidth, octaves, lacunarity, H);

    /* Create distorted clouds */
    PP = PP + Pdistortion;
    filtwidth = filterwidthp(PP);

    /* Compute fBm */
    result = fBm(PP, filtwidth, octaves, lacunarity, H);

    /* Adjust zero crossing (where the clouds disappear) */
    result = clamp(result+offset, 0, 1);

    /* Scale density */
    result /= (1 + offset);

    /* Modulate surface opacity by the cloud value */
    Oi = result * Os;

    /* Shade like matte, but with color scaled by cloud opacity */
    Ci = Oi * (Ka * ambient() + Kd * diffuse(faceforward(normalize(N),I)));
}
```

Team LRN

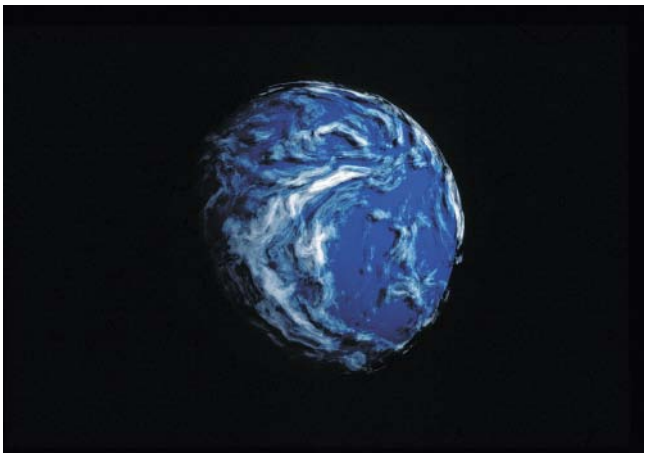


FIGURE 15.3 Clouds distorted with fBm: they look more like cotton than turbulence. Copyright © 1994 F. Kenton Musgrave.

Unfortunately, the result looks more like cotton than turbulence (see Figure 15.3), but it is an interesting experiment.

Once again, the essential element that our turbulence model lacks is vortices, or vorticity. Large-scale vortices in Earth's atmosphere occur in cyclonic and anticyclonic storm systems that are clearly visible from space. The most extreme vortices in our atmosphere occur in tornadoes and hurricanes. In Figure 15.4 we see an ontogenetic model of a hurricane, produced by the following specification:

```
surface
cyclone(float Ka = 0.5, Kd = 0.75;
        float max_radius = 1;
        float twist = 0.5;
        float scale = .7, offset = .5;
        float H = 0.675;
        float octaves = 4;)
{
    float radius, dist, angle, eye_weight, value;
    point Pt;             /* Point in texture space */
    vector PN;            /* Normalized vector in texture space */
    point PP;             /* Point after distortion */
    float filtwidth, a;

    /* Transform to texture coordinates */
    Pt = transform("shader", P);
    filtwidth = filterwidthp(Pt);
```

Team LRN

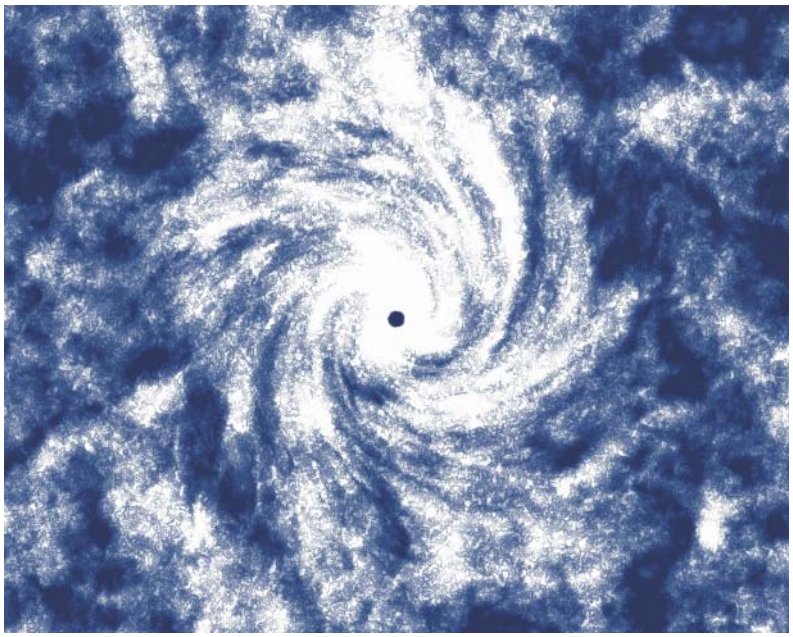


FIGURE 15.4 A first cut at including the vortices that comprise turbulence. Note that the smaller clouds are not distorted by the vortex twist, only the large-scale distribution is. Copyright © 1994 F. Kenton Musgrave.

```
/* Rotate hit point to "cyclone space" */
PN = normalize(vector Pt);
radius = sqrt(xcomp(PN)*xcomp(PN) + ycomp(PN)*ycomp(PN));

if (radius < max_radius) { /* inside of cyclone */
  /* invert distance from center */
  dist = pow(max_radius-radius, 3);
  angle = PI + twist * TWOPI * (max_radius-dist) / max_radius;
  PP = rotate(Pt, angle, point(0,0,0), point(0,0,1));
  /* Subtract out "eye" of storm */
  if (radius < 0.05*max_radius) { /* if in "eye" */
     eye_weight = ( . 1*max_radius-radius) * 10; /* normalize */
     /* invert and make nonlinear */
     eye_weight = pow(1 - eye_weight, 4);
  }
  else eye_weight = 1;
}
else { /* outside of cyclone */
  PP = Pt;
  eye_weight = 0;
}
```

Team LRN

```
    if (eye_weight > 0) { /* if in "storm" area */
      /* Compute clouds */
      a = DistfBm(PP, filtwidth, octaves, 2, H, 1);
      value = abs(eye_weight * (offset + scale * a));
    }
    else value = 0;

    /* Thin the density of the clouds */
    Oi = value * Os;

    /* Shade like matte, but with color scaled by cloud opacity */
    Ci = Oi * (Ka * ambient() + Kd * diffuse(faceforward(normalize(N),I)));
}
```

Here we have a single vortex; to model general turbulent flow we require a fractal hierarchy of vortices. Note that we've modeled the clouds on two different scales here: a distorted large-scale distribution comprising a weighting function, which is applied to smaller-scale cloud features. This construction is based on my subjective study of clouds and storms seen from space (see Kelley 1988 for many lovely examples of such images). The idea of undistorted small features corresponds to the phenomenon of viscosity, which, as our poem indicates, damps turbulence at small scales. This may also be seen as a first step in the direction of multifractal models, as our fractal behavior is different at different scales, and therefore may require more than one value or measure to characterize the fractal behavior.

The Coriolis Effect

A salient feature of atmospheric flow on Earth, and even more so on Venus, is that it is strongly affected by the *Coriolis effect*—a shearing effect caused by the conservation of angular momentum as air masses move north and south. The atmosphere is moving around the planet and, like a spinning skater who spins faster as she pulls her arms and legs in closer, the air must spin around the planet faster as it moves toward the poles and more slowly as it moves toward the equator. For a given angular momentum, angular velocity (spin rate) varies as the inverse square of radius. In this respect, the following model could be thought of as a physically accurate model, but I would still call it an ontogenetic, or at best an empirical, model.

Using our modified fBm and a Coriolis distortion, we can model Venus (as imaged at ultraviolet wavelengths) quite well (see Figure 15.5).

```
surface
venus(float Ka = 1, Kd = 1;
      float offset = 1;
```

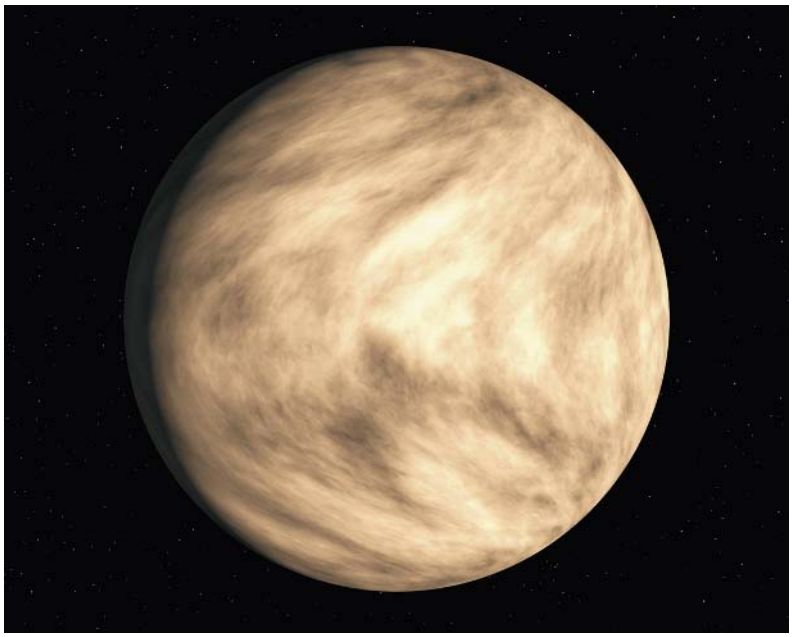


FIGURE 15.5 A strong Coriolis twist characterizes the cloud patterns on Venus. Copyright © 1994 F. Kenton Musgrave.

```
      float scale = 0.6;
      float twist = 0.22;
      float H = 0.65;
      float octaves = 8;)
{
   point Ptexture;   /* the shade point in texture space */
   vector PtN;       /* normalized version of Ptexture */
   point PP;         /* Point after rotation by Coriolis twist */
   float rsq;        /* Used in calculation of twist */
   float angle;      /* Twist angle */
   float value;      /* Fractal sum is stored here */
   float filtwidth;  /* Filter width for antialiasing */

   /* Transform to texture coordinates */
   Ptexture = transform("shader", P);
   filtwidth = filterwidthp(Ptexture);

   /* Calculate Coriolis twist distortion */
   PtN = normalize(vector Ptexture);
   rsq = xcomp(PtN)*xcomp(PtN) + ycomp(PtN)*ycomp(PtN);
```

Team LRN

```
   angle = twist * TWOPI * rsq;
   PP = rotate(Ptexture, angle, point(0,0,0), point(0,0,1));

   /* Compute Coriolis-distorted clouds */
   value = abs(offset + scale * fBm(PP,filtwidth,octaves,2,H));

   /* Shade like matte, but with color scaled by cloud color */
   Oi = Os;
   Ci = Oi * (value * Cs) *
        (Ka * ambient() + Kd *
        diffuse(faceforward(normalize(N),1)));
}
```

FIRE

Fire is yet another example of turbulent flow. I can't claim to have modeled fire particularly well, but I do have a fire texture that I can share with you. There are two peculiar features in this model. The first is the use of a "ridged" fBm, as discussed in the next chapter. The second is a distortion that varies exponentially with height, which is meant to model the upward acceleration of the hot gases of the flames.

```
surface
flame(float distortion = 0;
      float chaosscale = 1;
      float chaosoffset = 0;
      float octaves = 7;
      float flameheight = 1;
      float flameamplitude = .5; )
{
   point PP, PQ;
   float chaos;
   float cmap;
   float fw;

   PQ = PP = transform("shader", P);
   PQ *= point(1, 1, exp(-zcomp(PP)));
   fw = filterwidthp(PQ);

   chaos = DistfBm(PQ, fw, octaves, 2, 0.5, distortion);
   chaos = abs(chaosscale*chaos + chaosoffset);
   cmap = 0.85*chaos + 0.8 * (flameamplitude - flameheight * zcomp(PP);
   Ci = color spline(cmap,
                color(0, 0, 0),        color(0, 0, 0),
                color(27, 0, 0),       color(54, 0, 0),
                color(81, 0, 0),       color(109, 0, 0),
                color(136, 0, 0),      color(166, 5, 0),
                color(189, 30, 0),     color(211, 60, 0),
                color(231, 91, 0),     color(238, 128, 0),
```

Team LRN

```
                color(244, 162, 12),   color(248, 187, 58),
                color(251, 209, 115),  color(254, 236, 210),
                color(255, 241, 230),  color(255, 241, 230)) / 255;
}
```

While this certainly doesn't fully capture the rich structure of real flames, it's not too bad as a first approximation. In Figure 15.6 several transparent layers of this fire texture were sandwiched with some of Przemek Prusinkiewicz's early L-system tree models (Prusinkiewicz and Lindenmayer 1990). Note that the colors are critical to the realism of the results. What you want is a color map that represents a range of colors and intensities of black-body radiators, going from black to cherry red, through orange and yellow, to white.[4] Also, if you scale the flame structure down so that the high frequencies are more visible, this texture takes on a kind of astrophysical character—the ridges in the fBm form filaments and loops that, while not exactly realistic, have a distinctive character reminiscent of certain emission nebulae where stars are being born and dying. Try constructing this function without using `abs()` and observe the difference—it's actually more realistic and less surreal.

WATER

To my chagrin, people often say of my landscape images, "The water is the best part!" Well, I'm here to tell you that it's the cheapest and easiest trick I ever did with procedural textures. Let me show you how.

Noise Ripples

I originally developed this texture in a ray tracer that didn't have displacement mapping. In that context, it can be implemented as a bump map. This requires that you build a vector-valued fBm function, which is no problem as it's really just three independent fBms, one for each of *x*, *y*, and *z*, or one fBm constructed from `VecNoise()`, as we saw earlier in the definition of `VfBm()`. In MojoWorld and RenderMan, displacement mapping is available, so the implementation is simpler: it

4. Black-body radiators are a concept from physics: they are theoretically ideal (and therefore nonexistent in nature) objects that are completely without color of their own, regardless of their temperature. Thus, as you raise their temperature, they glow with a color that represents the ideal thermal, or Planck, spectrum for their temperature. Real objects heated enough to glow at visible wavelengths tend to be complicated both by their underlying (i.e., cold) nonblack color and by emission lines—certain wavelengths that have enhanced emission (or absorption) due to quantum effects. Most fires we see are made more yellow than a black-body radiator at the same temperature by sodium emission lines.

Team LRN

FIGURE 15.6 *Forest Fire* illustrates a fire texture. Color is critical in getting the look of fire. Note that the 1 − abs(noise) basis function gives rise to the sinews in the flames. Copyright © F. Kenton Musgrave.

requires only a scalar-valued displacement to perturb the surface, rather than a vector-valued perturbation for the surface normal.

The water model I've always used requires only a reflective silver plane in the foreground and this simple bump or displacement map applied to the surface:

```
displacement
ripples(float Km = 1, octaves = 2;)
{
   float offset;
   point PP;

   /* Do the calculations in shader space */
   PP = transform( "shader", P );

   /* Get fractal displacement, scale by Km */
   offset = Km * fBm( PP, 3.0, 2.0, octaves );
```

Team LRN

```
    /* Displace the surface point and recompute the normal */
    P += offset * normalize( N );
    N = calculatenormal( P );
}
```

We have perturbed the surface normal with a degenerate—in the mathematical sense—fractal function of only two octaves. (Recall that we said in the previous chapter that a fractal ought to have at least three levels of self-similarity to be called a fractal; this one only has two. Of course, you may use more or less octaves, as your artistry dictates.) You can stretch this function by scaling it up in one of the lateral directions, to get an impression of waves. For such a simple texture, it works surprisingly well (see Figures 15.7 and 15.8).

Wind-Blown Waters

In nature, ripples on water so calm are rarely homogeneous as those given by the previous function. Usually, on the large scale, ripples are modulated by the blowing

FIGURE 15.7 *Spirit Lake* shows a noise-based water bump map. The terrain model is the first multifractal model described in Chapter 16. Copyright © F. Kenton Musgrave.

Team LRN

FIGURE 15.8 *Lethe* is a polygon subdivision terrain model, hence the jagged appearance. It illustrates the water, sedimentary rock strata, and moon textures described in this chapter. Copyright © F. Kenton Musgrave.

wind. Blowing wind is turbulent flow and is therefore fractal. We can apply a large-scale, as compared to the ripples, fractal weighting function to the previous ripples to generate a nice approximation of breeze-blown water:

```
displacement
windywave(float Km = 0.1;
          float txtscale = 1;
          float windfreq = 0.5;
          float windamp = 1;
          float minwind = 0.3)
{
   float offset;
   point PP;
   float wind;
   float turb;
   float filtwidth;
```

```
    PP = txtscale * windfreq * transform("shader", P);
    filtwidth = filterwidthp(PP);

    /* get fractal displacement*/
    offset = Km * fBm(PP,filtwidth,2,2,0.5);
    PP *= 8; filtwidth *= 8;

    /* calculate wind field */
    turb = turbulence(PP, filtwidth, 4, 2, 0.5);
    wind = minwind + windamp * turb;

    /* displace the surface*/
    N = calculatenormal(P+wind * offset * normalize(N));
}
```

The weighting function we use is, yet again, a variety of fBm. We use the most generic version, as there's no need for the subtleties of any of the variations we've developed. (Turbulence is actually multifractal, however, so it might be worthwhile to experiment with multifractal weighting functions.) We need only a few octaves in this fBm, since we don't really want to generate fine-scale detail with the weighting function.

The result of the `Windywave()` texture is illustrated in Figures 15.9 and 15.18.

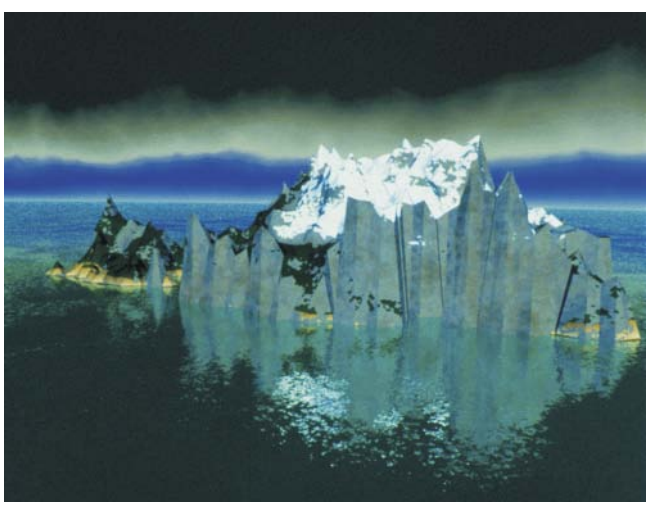


FIGURE 15.9 The effects of breezes on the water are illustrated in *Sea Rock*. Copyright © 1987 F. Kenton Musgrave.

Team LRN

EARTH

Now let's look at earthly textures. Well, at least two out of three will be literally earthly; the third will be a moon.

Sedimentary Rock Strata

Early in my career of rendering fractal landscapes, I didn't have the capacity to ray-trace terrains with very many triangles in them. Thus the "mountains" were quite chunky, and the triangles used to tessellate the surface were quite large and obviously flat, unlike any terrain I have ever seen. One of the early textures I devised to distract the eye's attention from this intrinsically bogus geometry was an imitation of sedimentary rock strata:

```
surface
strata(float Ka = 0.5, Kd = 1;
       float txtscale = 1;
       float zscale = 2;
       float turbscale = 0.1;
       float offset = 0;
       float octaves = 8;)
{
   color Ct;
   point PP;
   float cmap;
   float turb;
   PP = txtscale * transform("shader", P);

    /*turbation by fBm */
    turb = fBm(PP, filterwidthp(PP), octaves, 2, 0.5);

    /*use turb and z to index color map */
    cmap = zscale * zcomp(PP) + turbscale * turb - offset;
    Ct = color spline(mod(cmap, 1),
             color(166, 131, 70),   color(166, 131, 70),
             color(204, 178, 127),  color(184, 153, 97),
             color(140, 114, 51),   color(159, 123, 60),
             color(204, 178, 127),  color(230, 180, 80),
             color(192, 164, 110),  color(172, 139, 80),
             color(102, 76, 25),    color(166, 131, 70),
             color(201, 175, 124),  color(181, 150, 94),
             color(161, 125, 64),   color(177, 145, 87),
             color(170, 136, 77),   color(197, 170, 117),
             color(180, 100, 50),   color(175, 142, 84),
             color(197, 170, 117),  color(177, 145, 87),
```

```
            color(170, 136, 77),   color(186, 156, 100),
            color(166, 131, 70),   color(188, 159, 104),
            color(168, 134, 74),   color(159, 123, 60),
            color(195, 167, 114),  color(175, 142, 84),
            color(161, 125, 64),   color(197, 170, 117),
            color(177, 145, 87),   color(177, 145, 87)) / 255;

    /* Shade like matte, but with color scaled by cloud color and opacity */
    Oi = Os;
    Ci = Oi * Cs * Ct * (Ka * ambient() + Kd *
    diffuse(faceforward(normalize(N),I)));
}
```

The key idea here is to index a color lookup table by altitude[5] and to perturb that altitude index with a little fBm. The geologic analogs to this are soft-sediment deformation, in which layers of sediment are distorted before solidifying into sedimentary rock. It's closely related to Ken Perlin's famous *marble* texture.

The color lookup table is loaded with a color map that contains bands of color that you, the artist, deem appropriate for representing the different layers of rock. Both aesthetics and my observations of nature indicate that the colors of the various layers should be quite similar and subdued, with one or two layers that really stand out tossed in to provide visual interest. For an example of this, see the red and yellow bands in Figure 15.10.

Gaea: Building an Entire Planet

In fact, you can build an entire Earth-like planet with a single procedural texture. (We now call such planets MojoWorlds; this text was originally written some 10 years ago.) Not surprisingly, such an ambitious texture gets rather complex. And, of course, it is quite fractal. In fact, fractals are used in three ways in this texture: as a displacement map to provide continents, oceans, and a fractal coastline; as a perturbation to a climate-by-latitude color map (much like our earlier rock strata map) providing an interesting distribution of mountains, deserts, forests, and so on; and finally as a color perturbation, to ameliorate lack of detail in areas that would all be the same color because they share the same lookup table index.

Now let's see how such a complex procedural texture evolves, step by step. The first step in creating an earth is to create continents and oceans. This can be

5. In my original C implementation of these texture functions, the color maps are stored in 256-entry lookup tables. Larry Gritz has used RenderMan's functionality to replace those tables with splines in the code that appears in this text.

Team LRN

FIGURE 15.10 *Bryce* illustrates a sedimentary rock strata texture. Note the red and yellow strata that provide visual interest. The terrain model is a polygon subdivision erosion model described by Mandelbrot (Peitgen and Saupe 1988). Copyright © F. Kenton Musgrave.

accomplished by quantizing an fBm texture: A parameter threshold controls the "sea level"; any value less than threshold is considered to be below sea level and is therefore colored blue. This gives us the effect seen in Figure 15.11(a). Note that in the following code, there is a Boolean-valued parameter `multifractal`. This gives us the option of creating heterogeneous terrain and coastlines—see how the fractal dimension of the coasts varies in Figure 20.4.

Next we provide a color lookup table to simulate climatic zones by latitude; see Figure 15.11(b). Our goal is to have white polar caps and barren, gray sub-Arctic zones blending into green, temperate-zone forests, which in turn blend into buff-colored desert sands representing equatorial deserts. (Of course, you can use whatever colors you please.) The coloring is accomplished with a splined color ramp, indexed by the latitude of the ray/earth intersection point.

This rough coloring-by-latitude is then fractally perturbed, as in Figure 15.11(c). This is accomplished simply by adding fBm to the latitude value to perturb it, as with the rock strata texture. (This is just another example of the powerful tool of domain

Team LRN

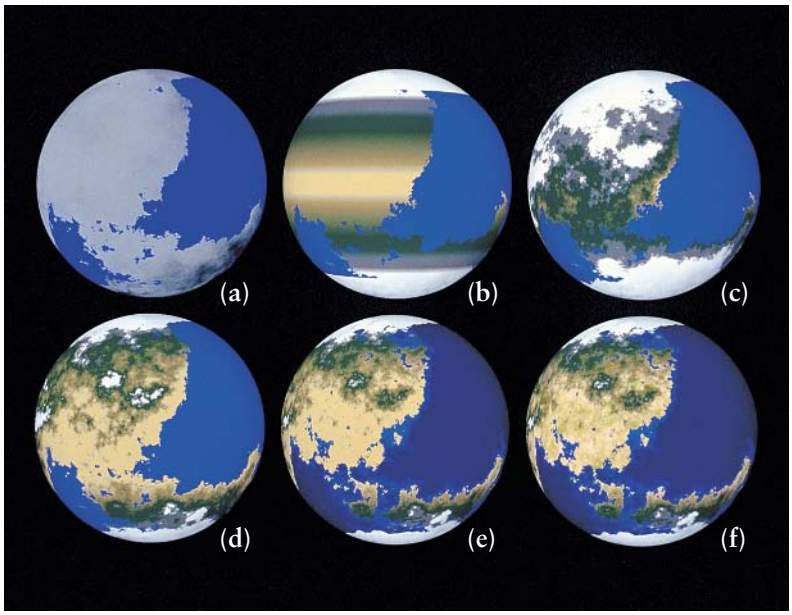


FIGURE 15.11 A sequence of stages in the development of a planetary structure. Copyright © F. Kenton Musgrave.

distortion.) We also take into account the displacement map, so that the "altitude" of the terrain may affect the climate. Note that altitude and latitude represent two independent quantities that could be used as parameters to a two-dimensional color map; to date I have used only a one-dimensional color spline for simplicity.

Next we add an exponentiation parameter to the color spline index computed earlier, to allow us to "drive back the glaciers" and expand the deserts to a favorable balance, as in Figure 15.11(d).

We now modify the oceans, adjusting the sea level for a pleasing coastline and making the color a function of "depth" to highlight shallow waters along the coastlines, as seen in Figure 15.11(e). Depth of the water is calculated in exactly the same way as the "altitude" of the mountains—as the magnitude of the bump vector in the direction of the surface normal. This depth value is used to darken the blue of the water almost to black in the deepest areas. It might also be desirable to modify the surface properties of the texture in the ocean areas, specifically the specular highlight, as this significantly affects the appearance of Earth from space (again, see Kelley 1988), although I haven't yet tried it.

Team LRN

Finally, we note that the "desert" areas about the equator in Figure 15.11(e) are quite flat and unrealistic in appearance. Earth, by contrast, features all manners of random mottling of color. By interpreting an fBm function as a color perturbation, we can add significantly to the realism of our model: compare Figure 15.11(e) and 15.11(f). This added realism is of an artistic nature—the color mottling of Earth does not, in general, resemble fBm—but it is nevertheless aesthetically effective.

This code accomplishes all of the above:

```
#define N_OFFSET 0.7
#define VERY_SMALL 0.0001
surface
terran(float Ka = .5, Kd = .7;
       float spectral_exp = 0.5;
       float lacunarity = 2, octaves = 7;
       float bump_scale = 0.07;
       float multifractal = 0;
       float dist_scale = .2;
       float offset = 0;
       float sea_level = 0;
       float mtn_scale = 1;
       float lat_scale = 0.95;
       float nonlinear = 0;
       float purt_scale = .9;
       float map_exp = 0;
       float ice_caps = 0.9;
       float depth_scale = 1;
       float depth_max = .5;
       float mottle_limit = 0.75;
       float mottle_scale = 20;
       float mottle_dim = .25;
       float mottle_mag = .02;)
{
   point PP, P2;
   vector PtN;
   float chaos, latitude, purt;
   color Ct;
   point Ptexture, tp;
   uniform float i;
   float o, weight;   /* Loop variables for fBm calculation */
   float bumpy;
   float filtwidth, fw;


   /* Do all shading in shader space */
   Ptexture = transform("shader", P);
   filtwidth = filterwidthp(Ptexture);
   PtN = normalize(vector Ptexture); /* Normalize Ptexture to radius 1.0 */
```

Team LRN

```
/*********************************************************************
* First, figure out where we are in relation to the oceans/mountains.
* Note: this section of code must be identical to "terranbump" if you
* expect these two shaders to work well together.
*********************************************************************/

if (multifractal == 0) {   /* use a "standard" fBm bump function */
   bumpy = fBm(Ptexture, filtwidth, octaves, lacunarity, spectral_exp);
} else {                   /* use a "multifractal" fBm bump function */
   /* get "distortion" vector, as used with clouds */
   Ptexture += dist_scale * filteredvsnoise(Ptexture, filtwidth);
   /* compute bump vector using MfBm with displaced point */
   o = spectral_exp; tp = Ptexture;
   fw = filtwidth;
   weight = abs(filteredDistNoise(tp, fw, 1.5));
   bumpy = weight * filteredsnoise(tp, fw);

   /* Construct a multifractal */
   for (i = 1; i < octaves && weight >= VERY_SMALL && fw < 1; i += 1) {
      tp *= lacunarity;
      fw *= filtwidth;
      /* get subsequent values, weighted by previous value */
      weight *= o * (N_OFFSET + snoise(tp));
      weight = clamp(abs(weight), 0, 1);
      bumpy += snoise(tp) * min(weight, spectral_exp);
      o *= spectral_exp;
   }
}

/* get the "height" of the bump, displacing by offset */
chaos = bumpy + offset;
/* set bump for land masses (i.e., areas above "sea level") */
if (chaos > sea_level) {
   chaos *= mtn_scale;
   P2 = P + (bump_scale * bumpy) * normalize(N);
} else P2 = P;
N = calculatenormal(P2);

/*********************************************************************
* Step 2: Assign a climate type, roughly by latitude.
*********************************************************************/

/* make climate symmetric about equator */
latitude = abs(zcomp(PtN));

/* Fractally purturb color map offset using "chaos"
 * "nonlinear" scales purturbation-by-z
 * "purt_scale" scales overall purturbation
 */
```

```
    latitude += chaos*(nonlinear*(1-latitude) + purt_scale);
    if (map_exp > 0 ) /* Perform nonlinear "driving the glaciers back" */
      latitude = lat_scale * pow(latitude,map_exp);
  else latitude *= lat_scale;

    if (chaos > sea_level) {
      /* Choose color of land based on the following spline.
       * Ken originally had a huge table. I was too lazy to type it in,
       * so I used a scanned photo of the real Earth to select some
       * suitable colors.—Larry Gritz
       */
      Ct = spline(latitude,
            color(.5, .39, .2),
            color(.5, .39, .2),
            color(.5, .39, .2),
            color(.2, .3, 0 ),
            color( .085, .2, .04),
            color(.065, .22, .04),
            color(.5, .42, .28),
            color(.6, .5, .23),
            color(1,1,1),
            color(1,1,1));

      /* Mottle the color to provide visual interest */
      if (latitude < mottle_limit) {
         PP = mottle_scale * Ptexture;
         purt = fBm(PP, mottle_scale*filtwidth, 6, 2, mottle_dim);
         Ct += (mottle_mag * purt) * (color(0.5, 0.175, 0.5));
      }
    }
    else {
      /* Oceans */
      Ct = color(.12, .3, .6);
      if (ice_caps > 0 && latitude > ice_caps)
         Ct = color(1,1,1); /* Ice color */
      else {
         /* Adjust color of water to darken deeper seas */
         chaos -= sea_level;
         chaos *= depth_scale;
         chaos = max(chaos, -depth_max);
         Ct *= (1+chaos);
      }
    }

    /* Shade using matte model */
    Ci = Ct * (Ka * ambient() + Kd * diffuse(faceforward(normalize(N),I)));
    Oi = Os; Ci *= Oi;
}
```

Selene

Now I'll show you an example of extreme hackery in pursuit of a specific visual end: my texture that can turn a featureless gray sphere into an imitation of the Moon, replete with lunar highlands, maria, and a single rayed crater.[6] I reasoned that one crater, if spectacular enough, would suffice to give an overall impression of moonliness. In retrospect, I guess it was, because this image caused Digital Domain to contact Larry and me for help in modeling the Moon for the movie *Apollo 13*. This in turn led to my being employed there, where I got to experience firsthand the madness and excitement of the Hollywood digital effects business, working on movies like *Titanic, Dante's Peak,* and *Air Force One*. Heady stuff—everyone should try it for a year or two!

Back to the moon texture: The highlands/maria part of the texture is just a simpler (in that it is not multifractal) variation on the continent/ocean part of the above texture. The rayed crater is the interesting part. It is a classic example of ontogenetic modeling taken to an extreme: the dynamics of such an impact and its resulting visual consequences would be very difficult to model physically, yet the phenomenon is essential to the appearance of the Moon. So I set out to construct a reasonable visual semblance through whatever chicanery I could devise. The resulting crater has two parts, which I originally implemented as two separate C functions: the bump-mapped crater rim and the rays, which are simply a surface color texture.

The crater bump map consists of a central peak, a common substrate-rebound feature seen in large impact craters; an unperturbed crater floor, which is resurfaced with smooth lava in the real craters; and a ring that delineates the edge of the crater. The ring is designed to have a steep slope on the inside and a more gradual one on the outside, again in emulation of the structure of real impact craters. Furthermore, the outside ring has a texture of random ridges, concentric with the crater. This is in emulation of compression features in the crust around the edge of the crater, formed when the shock of impact "bulldozed" the crater out from the center. We obtain this texture with radially compressed fBm. The composite result of these features is a detailed, fairly realistic model of a crater. Since it is applied as a bump map, as with ray tracers, you must be careful not to try to view the crater at a very low, glancing

6. If you've never seen a rayed crater before, just grab a pair of binoculars next time there's a full moon, and have a look. They're all over the place, they're not particularly subtle, and some have rays that reach more than halfway around the globe of the Moon. The rays are the result of splattering of ejecta from the impact that made the crater.

Team LRN

angle, for then the lack of geometric detail becomes evident. A problem with bump maps is that since they do not affect geometry, bump-mapped features cannot obscure one another, as the raised near rim of a real crater would obscure your view of the far rim at low angles of view. If your renderer supports displacement maps or QAEB primitives, you can model the geometry correctly.

The crater ray construction is inspired by the famous Doc Edgerton photo of the splashing milk drop. (You know, the one where it looks like a crown, with all the little droplets evenly spaced around the rim of the splash.) This high degree of regularity inspired me to build my texture in a similar way: with a number of rays, evenly spaced, with only a small random displacement in their spacing. The rays take the form of weighting functions, the width of which asymptotically approaches zero as we move out along the length of the ray. There is a discontinuity in the weighting function between rays, but this has never been visible in practice. The weighting is applied to an fBm splatter texture, which is stretched out *away* from the crater, exactly opposite of the compression texture outside the rim of the crater.

This crater ray texture looked much too regular to be realistic. So next I tried a fractal (fBm) splatter for the rays; the result was much too irregular. It seems that the behavior of nature lies somewhere between the two. I eventually settled on a combination of both: I use fractal splatter for the short-range ejecta and the ray scheme for the long-range ejecta. The combination looks reasonably good, although I think it would benefit from further, artful randomization of the ray structure. At the time I developed this model, I didn't need to inspect the resulting moon up close—it was designed to serve as a backdrop for earthly scenes—so I called it "good enough" and moved on, only slightly embarrassed to have put so much time and effort into such an entirely ad hoc model.

The code for the lunar texture follows. Figures 15.12 and 20.4 illustrate what it can look like.

```
surface
luna (float Ka = .5, Kd = 1;
      float lacunarity = 2, octaves = 8, H = 0.7;
      color highland_color = .7;
      float maria_basecolor = .7, maria_color = .1;
      float highland_threshold = -0.2;
      float highland_altitude = 0.01, maria_altitude = 0.004;
      float peak_rad = .0075, inner_rad = .01, rim_rad = .02, outer_rad = .05;
      float peak_ht = 0.005, rim_ht = 0.003;)
{
   float radial_dist;
   point PP;
   float chaos;
   color Ct;
```

Team LRN

FIGURE 15.12 The rayed crater is prominent on this procedurally textured moon. Copyright © 1994 F. Kenton Musgrave.

```
float temp;
point vv;
float uu, ht;
float lighten;
point NN;
float pd; /* pole distance */
float raydist;
float filtwidth;
float omega;
PQ = P;

PP = transform("shader", P);
filtwidth = filterwidthp(PP);
NN = normalize(N);

radial_dist = sqrt(xcomp(PP)*xcomp(PP) + ycomp(PP)*ycomp(PP));
omega = pow(lacunarity, (-.5)-H);
chaos = fBm(PP, filtwidth, octaves, lacunarity, omega);

/*** Get overall maria/highlands texture ************************/

chaos = fBm (PP, H, lacunarity, octaves);
```

Team LRN

```
/* Start out with the surface color, then modify as needed */
Ct = Cs;

/* Ensure that the crater is in one of the maria */
temp = radial_dist;
if (temp < 1)
   chaos -= .3 * (1 - smoothstep(0, 1, temp));

/* Determine highlands and maria */
if (chaos > highland_threshold) {
   PQ += chaos * highland_altitude * NN;
   Ct += highland_color * chaos;
} else {
   PQ += chaos * maria_altitude * NN;
   Ct *= maria_basecolor + maria_color * chaos;
}

/*** Add crater ************************************************/

pd = 1-v;
vv = vector(xcomp(PP)/radial_dist, 0, zcomp(PP)/radial_dist);
lighten = 0;
if (pd < peak_rad) {                    /* central peak */
   uu = 1 - pd/peak_rad;
   ht = peak_ht * smoothstep(0, 1, uu);
} else if (pd < inner_rad) {            /* crater floor */
   ht = 0;
} else if (pd < rim_rad) {              /* inner rim */
   uu = (pd-inner_rad) / (rim_rad - inner_rad);
   lighten = .75*uu;
   ht = rim_ht * smoothstep(0, 1, uu);
} else if (pd < outer_rad) {            /* outer rim */
   uu = 1 - (pd-rim_rad) / (outer_rad-rim_rad);
   lighten = .75*uu*uu;
   ht = rim_ht * smoothstep(0, 1, uu*uu);
} else ht = 0;

/* Lighten the higher areas */
PQ += ht * NN;
lighten *= 0.2;
Ct += color(lighten,lighten,lighten);

/* Add some noise to simulate rough features */
if (uu > 0) {
   if (pd < peak_rad) {                 /* if on central peak */
   vv = 5*PP + 3 * vv;
   ht = fBm(vv, filterwidthp(vv), 4, 2, 0.833);
   PQ += 0.0025 * uu*ht * NN;
 } else {
   vv = 6*PP + 3 * vv;
```

Team LRN

```
      ht = fBm(vv, filterwidthp(vv), 4, 2, 0.833);
      if (radial_dist > rim_rad) uu *= uu;
      PQ += 0.0025 * (0.5*uu + 0.5*ht) * NN;
    }
   }

   /*** Generate crater rays ***************************************/

   lighten = 0;
   if (pd >= rim_rad && pd < 0.4) {
      float fw = filterwidth(u);
      lighten = smoothstep(.15, .5, filteredsnoise(62*u, 62*fw) );
      raydist = 0.2 + 0.2 * filteredsnoise(20 * mod(u+0.022,1), 20*u);
      lighten *= (1 - smoothstep(raydist-.2, raydist, pd));
   }
   lighten =0.2 * clamp(lighten, 0, 1);
   Ct += color(lighten, lighten, lighten);

   /* Shade like matte */
   Ci = Ct * (Ka * ambient() + Kd * diffuse(faceforward(normalize(N),I)));
   Oi = Os; Ci *= Oi;
}
```

RANDOM COLORING METHODS

Good painters evoke worlds of color in a painting, and even in a single brush stroke. Van Gogh, who painted with a palette knife, not a brush, executed several paintings in the morning, took a long lunch, then did several more in the afternoon. Painting in a hurry, he didn't mix his paints thoroughly before applying a thick blob to the canvas. Thus each stroke has a universe of swirling color within it. Seurat's pointillism is another form of what painters call *juxtaposition,* or the use of a lot of different colors to average to some other color. This is part of the visual complexity that, to me, distinguishes good paintings from most computer graphics.

Generating visual complexity is pretty much the name of my game, so I've devised some methods for procedurally generating complexity in color. They utilize the same kinds of fractal functions that we use in modeling natural phenomena.

Random fBm Coloring

My first attempt at random coloring was simply fBm interpreted as color. For this, you start with a vector-valued fBm that returns three values and interpret that 3-vector as an RGB color. This can work pretty well, as seen in Figure 20.9. Unfortunately, since fBm has a Gaussian distribution with an expected value of zero, when one of the values (say, red) is fairly far from zero, the other two are likely to be close

to zero. This yields a preponderance of red, green, and blue blotches. We want more control and color variation than that.

The GIT Texturing System

I wanted to obtain a rich, fractal variation in color detail, similar to the juxtaposition in a van Gogh stroke or a local area in a Seurat. This juxtaposition should average to a user-specified color, even though that color may not be present anywhere in the resulting palette. We also want easy user control of the color variation in the juxtaposition palette. Control of this palette is accomplished by manipulating the values of a 4 × 4 transformation matrix, but this is too mathematically abstract to constitute an effective user interface (UI), to say the least. We desire a simple and intuitive UI that an artist can use effectively, without knowledge of the underlying math.

Long ago I gave this idea the wonderfully unpretentious—not!—moniker "generalized Impressionistic texture," or GIT for short. (We need more TLAs—three-letter acronyms.) The GIT matrix generator system takes the form of a time-varying swarm of color samples in a color space, usually the RGB color cube. The center of the swarm is translated (moved) to the position in the color space of the desired average color of the resulting palette. A swarm of color sample chips is then manipulated to obtain the desired variation in color. This is accomplished by rotating and scaling the principal axes of color variation within the color space.[7] If you think of the scattering of color samples as lying within an ellipsoid, or an elongated M&M, the principal axes correspond to the length, width, and thickness of the M&M. After some manipulation, you might have a major axis of variation along, for instance, blue-to-yellow, with less significant variations along two other perpendicular axes in the chosen color space. The idea is that you can have a lot of different colors present, with their average clearly specified and smooth interpolation between them well defined.[8] The major variations can be along any straight line in color space, not just red, green, or blue; this is the flexibility of the system.

The simplest underlying mathematical model requires that the three axes of color variation be mutually perpendicular. In this model we build a standard 4 × 4

7. Which color space you use is important, as it affects the character of available color variations. To date, we've only implemented the RGB color cube. It is a little more challenging to display other color spaces as polyhedra, due to their nonrectilinear shapes. It shouldn't be too hard to do, though.

8. The average isn't quite as clear as it may seem, as the ellipsoid can violate the bounds of the color space. You can handle this by either clipping to the boundaries or reflecting them back into the color space (the solution we've used). Either solution will skew the average color away from that at the center of the ellipsoid, which marks the presumed average. There's no easy fix for this.

Team LRN

transformation matrix encoding rotation, translation, and scaling. The transformation matrix is built using interactive controls that let the user manipulate the scatter plot of color samples in the three-dimensional color space. In our implementation, the scatter plot is constantly changing: it is a circular queue of vector values representing offsets from the center of the ellipsoid, which represents the average color. When a sample reaches the end of the queue, it is replaced with another random sample. The random samples are gotten by evaluating vector-valued fBm at random points in its three-dimensional domain. We use about 100 samples at a time, the actual number being controlled by a slider. More samples span the range of colors more accurately, but the ones in front tend to obscure those behind, so you can't see the entire range of colors being spanned. The idea behind the circular queue is that by having the random samples constantly changing with time, you can get a pretty good idea of the range of colors spanned just by watching for a while.

When the desired distribution of color variations has been determined interactively, the resulting transformation matrix is used by a texture routine in a renderer. The result is a procedural solid texture with wonderfully rich variations in color. The colors of the mountains in Figures 15.13, 15.14, 16.6, 20.18, and 20.20 come from such GIT textures. I find them very pleasing because they finally start to capture in synthetic imagery some of the color complexity and subtlety we see in paintings.

An Impressionistic Image Processing Filter

The GIT scheme generates solid textures that are usually applied to the surfaces of objects in a scene. In painting, juxtaposition is in image space—on the canvas—not in world space or object space, as with solid textures. Hence we, Myeong Lim and myself, sought to apply GIT texturing in image space to digital images. In this case, the matrix is determined by performing *principal components analysis* (Gonzalez and Woods 1992) to local areas in the image.[9] In principal components analysis, the Hotelling transform is applied to a scattering of data, yielding the *autocorrelation matrix* for the distribution. This matrix encodes some magic values known mathematically as the *eigenvectors,* which correspond to the principal axes described earlier, and the *eigenvalues,* which correspond to the length of those axes. More mathematical magic! The data points we provide to the Hotelling transform are the RGB values of pixels, ranging from a fairly small neighborhood around a given pixel to the entire image.

9. Explaining this is well beyond the scope of this book; see Gonzalez and Woods (1992) for details if you're interested. But be advised, it involves some pretty heavy linear algebra and statistics.

Team LRN

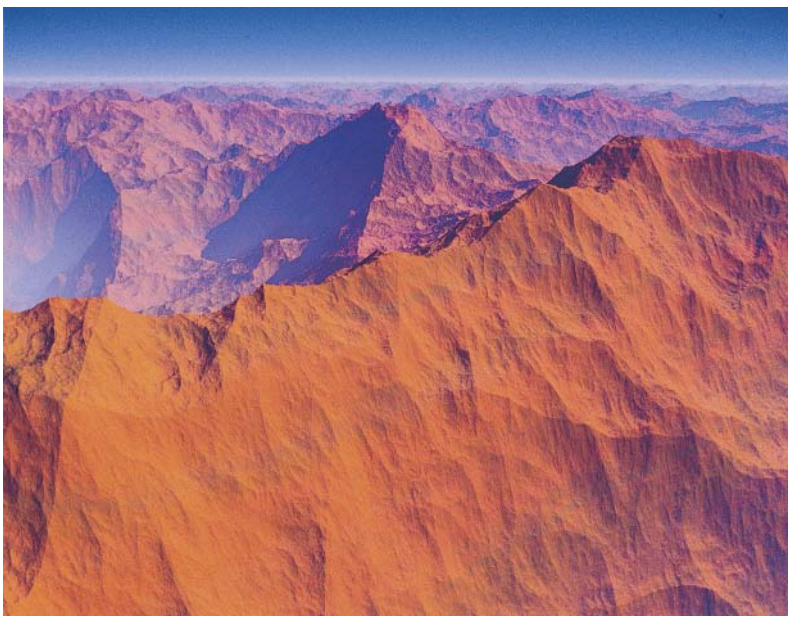


FIGURE 15.13 *Slickrock I* features a procedural terrain with adaptive level of detail and a terrain model constructed from a ridged basis function—hence the ridges everywhere and at all scales. The same subtle color perturbation has been applied to the surface as in Figure 20.9, but it has been "squashed down" vertically to make it resemble sedimentary rock strata. Copyright © F. Kenton Musgrave.

The autocorrelation matrix encodes exactly the same information as the interactively derived matrix in the GIT scheme. We use it in a similar, but slightly different, way: the juxtaposition is now expressed in a synthetic brush stroke. This is applied to the image as a blurring, yet detail-adding filter. The added detail is in the smeared colors in the synthetic brush strokes. To accommodate this added detail, we generally expand the image by a factor of four to eight in both dimensions. Standard image processing routines are used to determine lightness gradients in the input image, and the brush strokes are applied cross-gradient (Salisbury et al. 1994) to resemble an artist's strokes. Thus, if the image faded from dark at the bottom to light at the top, the strokes would be horizontal. While blurring the image underneath along the direction of the stroke (Cabral and Leedom 1993), random detail is simultaneously added in the form of fractal color juxtaposition. Figure 15.15 shows this in practice. This application of the GIT idea is probably less successful than commercially available paint programs such as Painter, but it was an interesting experiment. We never spent much time on the model for the brush strokes; this scheme could provide an interesting filter if the details were worked out.

Team LRN

FIGURE 15.14 *Slickrock III* illustrates a GIT texture applied to a terrain. Both the terrain and the texture feature adaptive level of detail, as described in Chapter 17. Copyright © F. Kenton Musgrave.

Team LRN

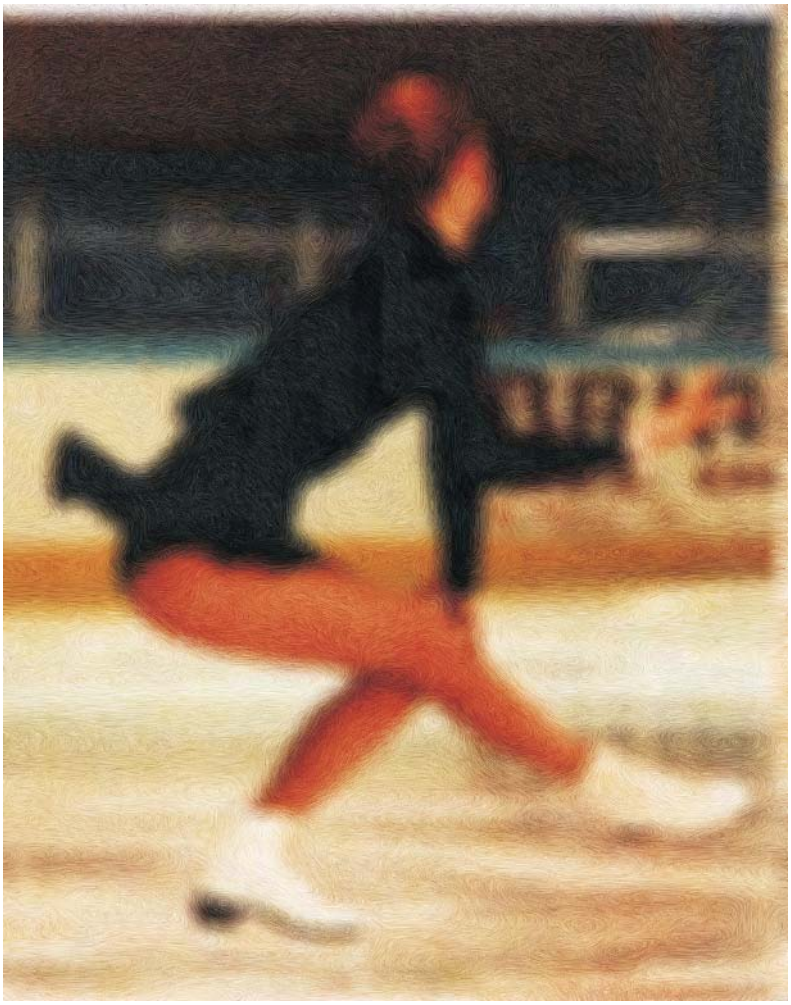


FIGURE 15.15 A GIT-processed image of Beth Musgrave skating, by Myeong Lim. The GIT processing adds random colors and the painterly quality. Copyright © F. Kenton Musgrave.

The "Multicolor" Texture

I always like to automate things as much as possible to see what the computer can be made to do on its own.[10] Being interested in the kind of painterly textures that the GIT experiments were designed to create, I set out to design a "painterly" procedural texture that is entirely random (see Figure 15.16). The result is, I think, rather

10. My considered view of the computer's role in generating art is that it is like an idiot savant assistant: it is extremely simple-minded, incapable of doing anything without the most exhaustively precise directions, but fantastically quick in what it does well—calculations, if you will. Its power in this can lead to fabulous serendipity, as we will see in Chapter 19, but the computer is never truly creative. All creativity resides in the human operator. While the computer can sometimes *appear* to be fabulously creative, it is an illusion, at least in these schemes.

Team LRN

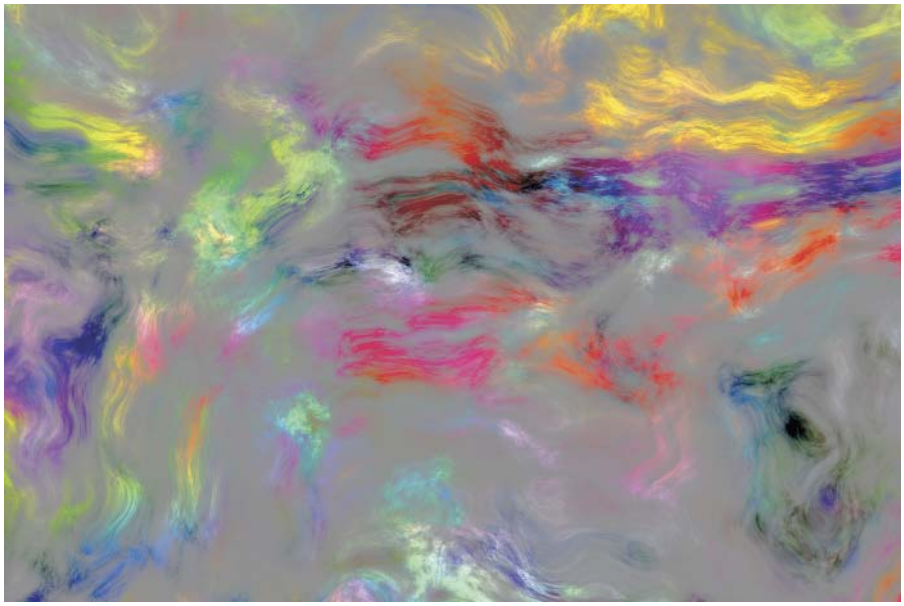


FIGURE 15.16 The "multicolor" texture attempts to capture some of the richness of color juxtaposition seen in paintings. Copyright © F. Kenton Musgrave.

striking, so I'll describe it here. It's a solution looking for a problem—I haven't found a way to include it in any images—but perhaps someday someone will find a use for it. It's also a good example of how a hacked-together texture can become brittle: small changes in the input parameters can "break" the result (meaning, your beautiful texture gets ugly).

The idea here is to generate a rich, painterly combination of random colors in a texture that looks something like flowing paint. Furthermore, we want the color to average to neutral gray. (It can actually average to any color, but I like the neutrality of 50% gray.) There are four basic elements in this texture: first, a vector-valued fBm that provides a random axis about which to rotate in color space; second, another vector-valued fBm provides a random color sample; third, a domain-distorted multifractal function modulates saturation in the random colors; and fourth, a rotation by a random amount around the random axis decorrelates the color from the RGB axes.

Let's go over this step by step. We start with a neutral gray plane. Then we get a random rotation axis from a vector-valued fBm function and store it for later use. We modulate its saturation with a multifractal function similar to the one rendered as a height field in Figure 14.3. Where that function is zero, the gray is unchanged. Where it's positive or negative, the gray becomes colored. Again, the color comes from a vector-valued fBm. As noted earlier, interpreting such an fBm vector as an RGB color doesn't give us the truly random variety of colors we want. We can get that using a random variation of the GIT scheme: simply build a random rotation matrix, corresponding to a GIT matrix, but without any translation or scaling. (The translation is inherent in the length of the fBm vector; the scaling is done by the

Team LRN

multifractal.) This may seem elaborate, but it has been my experience that you have to mess around with random colors a lot to get them to be truly random and to vary in a way that is pleasing to the eye.

Here's the shader code for "multicolor." As it (and some of the other textures in this chapter) is brittle, you might also want to look at the code for the original C version that appears on this book's Web site (*www.mkp.com/tm3*).

```
vector vMultifractalFunc(point p; float H, lacunarity, octaves, zero_off-set)
{
   point pos = p;
   float f = 1, i;

   vector y = 1;
   for (i = 0; i < octaves; i += 1) {
      y *= zero_offset + f * (2*vector noise(pos) - 1);
      f *= H;
      pos *= lacunarity;
   }
   return y;
}

/*
 * A multifractal multicolor texture experiment
 */
surface multicolor()
{
   vector axis, cvec, angle;
   point tx = transform("shader", P);
   float i;

   axis = 4.0 * vfBm(tx, filterwidthp(tx), 8, 2.0, 0.5);
   cvec = .5 * 5.0 * vfBm(tx*0.3, filterwidthp(tx), 7, 2.0, 0.5);
   tx += cvec;

   cvec += 4.0e5 * vMultifractalFunc(tx, 0.7, 2.0, 8, 0.2);
   angle = fBm(tx, filterwidthp(tx), 6, 2.0, 0.5);
   cvec = rotate(cvec, angle, point(0,0,0), axis);

   Ci = 0.5 + color(xcomp(cvec), ycomp(cvec), zcomp(cvec));

   /* Clamp color values to range [0..1] */
   for (i = 0; i<3; i += 1) {
      float c = abs(comp(Ci,i));
      if (c > 1)
      setcomp(Ci, i, 1-c);
   }
   Oi = Os;
   Ci *= Oi;
}
```

Team LRN

Ultimately, I think of this texture and the genetic textures I present in Chapter 19 as applications of the "naturalness" of the fractal functions developed earlier in this text, in pursuit of the look and feel of paintings, which I think of as being very "natural." That is, they may be man-made, but they appear very natural compared to the hard-edged artificiality of most synthetic images. The flow of the paint and the hand of the painter are very natural, and paintings are executed in a physical medium, rather than shuffling bits around for later output on some arbitrary device. This physicality is completely natural, compared to the abstractions of image synthesis. So I, personally, think of paintings as being a part of nature, as compared to what we're doing here. At any rate, I think of painters as the ancient masters of rendering, so I'm trying to learn from them by imitation. I find it fun and, when I get a good result, satisfying.

The "multicolor" texture provided the starting point for my genetic texture program described in Chapter 19. Development of this genetic program was driven partly by my desire to automate the generation of textures like "multicolor"—brittle textures like this are a real pain to design and refine—and partly by the knowledge that someday we're going to want to populate an entire virtual universe with procedurally generated planets, which, as we saw in this chapter, is also too much work to do on a planet-by-planet basis. You'll also see then that some of the fundamental functionality in my genetic program derives from the ideas I've described here about random coloration. At any rate, I think that the constructions in this chapter are mostly rather unique and nonobvious, since they involve a cross-fertilization of ideas from aesthetics and mathematics. I think that's cool. And best of all, they can make nice pictures.

PLANETARY RINGS

Rings like Saturn's are particularly easy to model (see Figures 15.17 and 15.18). You can simply employ a fractal function, evaluated as a function of radius, to control the transparency—and color, if you like—of a disk passing through the equator of the planet. The following C code is what was used to create Figure 15.18.

```
void Rings ( Vector intersect,
             double *refract,
             Color *color,
             double inner_rad, outer_rad,
             double f_dim, octaves, density_scale, offset,
             double inner_rolloff, outer_rolloff, /* in fraction of ring width */
             double text_scale, tx_offset, ty_offset, tz_offset )
{
```

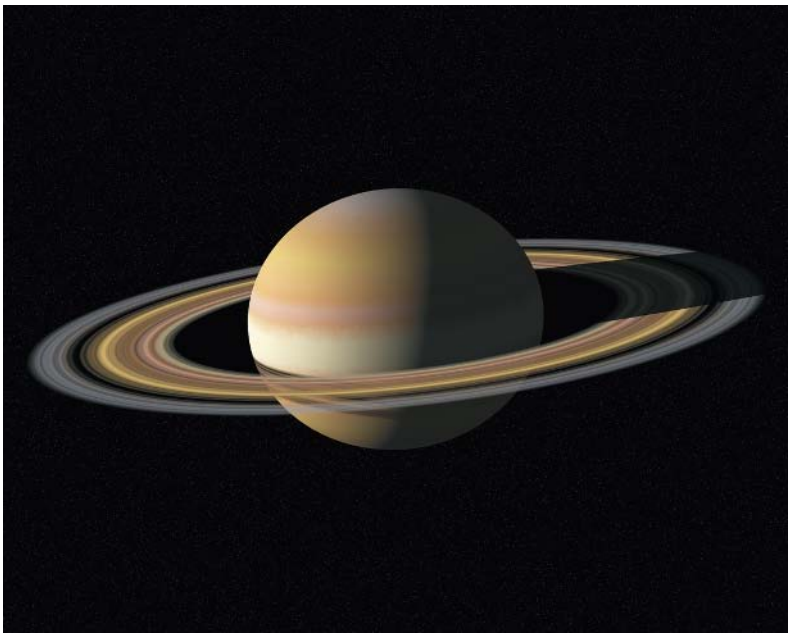


FIGURE 15.17 Planetary rings like Saturn's are easy to model by modulating density with a fractal function of radius.

FIGURE 15.18 Other style features a subtle use of the windywave shader and the model of Saturn seen in Figure 15.17.

```
      Vector  point;
      double  x_pos, y_pos, z_pos, radius, density, fBm();
      double  band_width, inner_rolloff_end, outer_rolloff_start, rolloff;


      x_pos = intersect.x;
      y_pos = intersect.y;
      z_pos = intersect.z;
      radius = sqrt (x_pos*x_pos + y_pos*y_pos + z_pos*z_pos);


      /* most of the work is in computing the inner and outer rolloff zones */
      if ( radius<inner_rad || radius>outer_rad ) {
            density = 0.0;
      } else {
            /* compute fBm texture */
            point = Vector (text_scale*radius + tx_offset, ty_offset, tz_offset);
            density = fBm( point, f_dim, 2.0, octaves );
            density = density_scale*density + offset;


            /* check for inner & outer rolloff zones */
            band_width = outer_rad - inner_rad;
            inner_rolloff_end =   inner_rad + band_width*inner_rolloff;
            outer_rolloff_start = outer_rad - band_width*outer_rolloff;
            if ( radius < inner_rolloff_end ) {
                  /* get rolloff parameter in range [0..1] */
                  rolloff = (radius-inner_rad)/(band_width*inner_rolloff);
                  /* give rolloff desired curve */
                  rolloff = 1.0 - rolloff;
                  rolloff = 1.0 - rolloff*rolloff;
                  /* do the rolling-off */
                  density *= rolloff;
            } else if ( radius > outer_rolloff_start ) {
                  /* get rolloff parameter in range [0..1] */
                  rolloff = (outer_rad-radius)/(band_width*outer_rolloff);
                  /* give rolloff desired curve */
                  rolloff = 1.0 - rolloff;
                  rolloff = 1.0 - rolloff*rolloff;
                  /* do the rolling-off */
                  density *= rolloff;
            }
      }
      /* clamp max & min values */
      if ( density < 0.0) density = 0.0;
      if ( density > 1.0) density = 1.0;


      transparency = 1.0 - density;

} /* Rings() */
```

16

Team LRN

PROCEDURAL FRACTAL TERRAINS

F. KENTON MUSGRAVE

As pointed out in Chapter 14, the same procedural constructions that we use as textures can also be used to create terrains. The only difference is that, instead of interpreting what the function returns as a color or other surface attribute, we interpret it as an altitude. This chapter will now extend the discussion of terrain models begun in Chapter 14.

Since we are designing these functions to generate terrain models external to the renderer or as QAEB primitives built into the renderer, we're switching back from the RenderMan shading language to C code for the code examples.

ADVANTAGES OF POINT EVALUATION

I first started working with procedural textures when I used them to color fractal terrains and to provide a sense of "environment," with clouds, water, and moons, as described earlier. Figure 15.8 is an example of this early work. The mountains I was making then were created with a version of polygon subdivision (hexagon subdivision) described by Mandelbrot in an appendix of *The Science of Fractal Images* (Peitgen and Saupe 1988). They have the jagged character of polygon subdivision terrains and the same-roughness-everywhere character of a homogeneous fractal dimension. Mandelbrot and I were working together at the time on including erosion features in our terrains. This led me to make some conjectures about varying the local behaviors of the terrain, which led to the two multifractal constructions I will describe next. Interestingly, I had never heard of "multifractals" when I devised these first two additive/multiplicative hybrid multifractal functions. When I showed Mandelbrot Figure 16.1 in 1991, he exclaimed in surprise, "A multifractal!" to which I astutely replied, "What's a multifractal?"[1]

What allowed me to intuitively "reinvent the (multifractal) wheel" was the flexibility implicit in our noise-based procedural fractals. Dietmar Saupe calls our Perlin

1. His cryptic retort was, "Never mind—now is not the time."

Team LRN

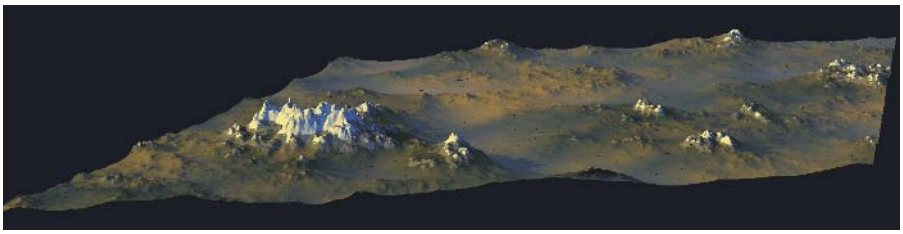


FIGURE 16.1 A multifractal terrain patch. Note the heterogeneity: plains, foothills, and mountains, all captured in a single fractal model. Copyright © 1994 F. Kenton Musgrave.

noise-based procedural fractal construction method "rescale and add" (Saupe 1989). Its distinguishing feature, he points out, is point evaluation: the fact that each sample is evaluated at a point in space, without reference to any of its neighbors. This is quite a distinction indeed, in the context of fractal terrain generation algorithms. In polygon subdivision, a given altitude is determined by a series of interpolations between neighboring points at lower frequencies (i.e., earlier steps in the iterative construction). In Fourier synthesis, the entire terrain patch must be generated all at once; no sample can be computed in isolation. In contrast, the context independence of our procedural method allows us to do whatever we please at any given point, without reference to its neighbors. There *is* interpolation involved, but it has been hidden inside the noise function, where it takes the form of Hermite spline interpolation of the gradient values at the integer lattice points (see Chapters 2, 6, and 7 for details on this). In practice, you could employ the same tricks described below to get multifractals from a polygon subdivision scheme, at least. It's not so obvious how you could accomplish similar effects with Fourier synthesis. The point is, I probably never would have thought of these multifractal constructions had I not been working in the procedural idiom.

Another distinguishing feature of terrains constructed from the noise function is that they can be rounded, like foothills or ancient mountains (see Figure 16.2). To obtain this kind of morphology from polygon subdivision, we must resort to the complexities of schemes like Lewis's "generalized stochastic subdivision" (Lewis 1987). The rounded nature of our terrains has to do with the character of the basis function; more on that later. And, as we have already shown, another distinguishing characteristic of the procedural approach is that it naturally accommodates adaptive band-limiting of spatial frequencies in the geometry of the terrain as required for rendering with adaptive level of detail, as in QAEB rendering (described in the next

Team LRN

FIGURE 16.2 *Carolina* illustrates a procedural model of ancient, heavily eroded mountains. Copyright © 1994 F. Kenton Musgrave.

chapter). Such capability makes possible exciting applications like the planetary zoom seen in Figure 16.3.

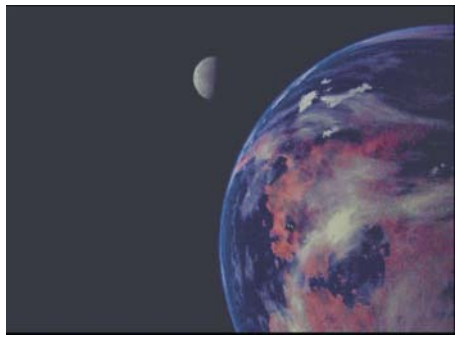
THE HEIGHT FIELD

Terrain models in computer graphics generally take the form of *height fields*. A height field is a two-dimensional array of altitude values at regular intervals (or *post spacings*, as geographers call them). So it's like a piece of graph paper, with altitude values stored at every point where the lines cross.[2]

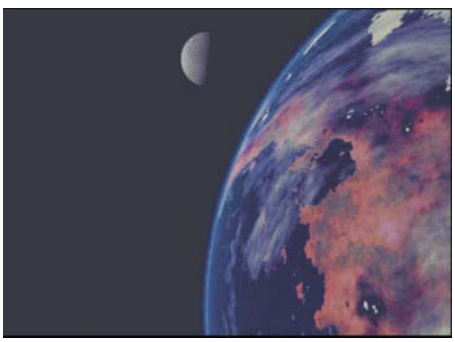
2. This is the simplest, but not the most efficient, storage scheme for terrains. Extended areas of nearly constant slope can be represented with fewer samples, for instance. Decimation algorithms (Schroeder, Zarge, and Lorensen 1992) are one way to reduce the number of samples in a height field and thus its storage space requirement. It may be desirable to resample such a model before rendering, however, to get back to the regular array of samples that facilitates fast rendering schemes such as grid tracing. For an animation of this zoom, see *www.kenmusgrave.com/animations.html*.

Team LRN

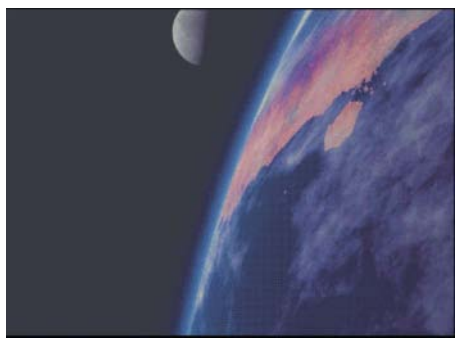
(a)



(b)



(c)

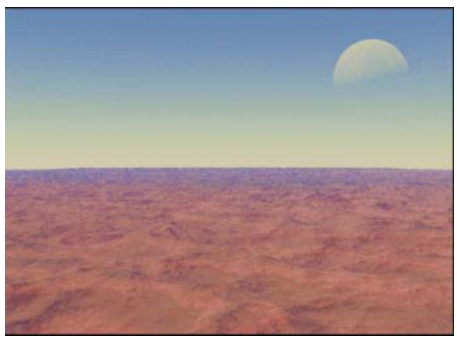


(d)

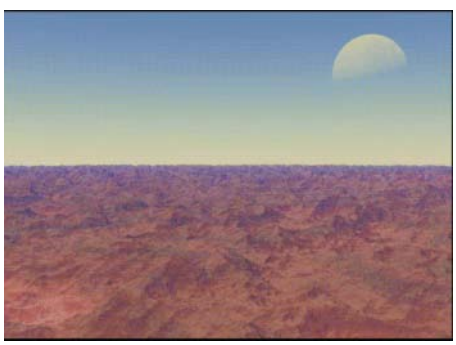
(e)

FIGURE 16.3 A series of frames from the *Gaea Zoom* animation demonstrate the kind of continuous adaptive level of detail possible with the models presented in the text. The MPEG animation is available at *www.kenmusgrave.com/animations.html*. Copyright © F. Kenton Musgrave.

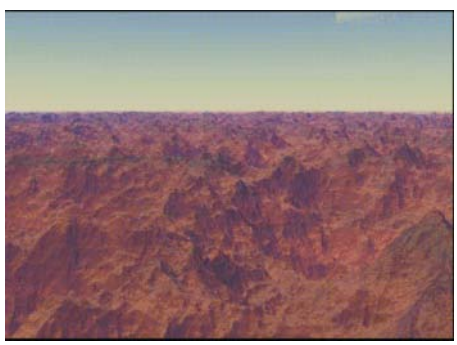
(f)



(g)



(h)



(i)

There is one, and only one, altitude value at each grid point. Thus there can be no caves or overhangs in a height field. This limitation may seem easy to overcome, but it's not. In fact, the problem is hard enough that I issued a challenge in the first edition of this book, back in 1994, offering $100 to the first person to come up with an "elegant, general solution" to it. The reward was won in early 1998 by Manuel Gamito, who came over from Portugul to work with us in the MetaCreations Skunk Works for a year. Manuel's award-winning image, an entirely procedural model rendered in a modified version of the minimal ray tracer I wrote to develop the QAEB algorithm, appears in Figure 17.2. Unfortunately, that image took about a day to render!

Team LRN

The regular (i.e., evenly spaced) samples of the height field accommodate efficient ray-tracing schemes such as *grid tracing* (Musgrave 1988) and quad tree (Kajiya 1983a) spatial subdivision. A detailed discussion of such a rendering scheme is beyond the scope of this book; if you're interested in that, see Musgrave (1993). I've always preferred to ray-trace my landscapes, but if you lack the computational horsepower for that, there are some very nice non-ray-tracing terrain renderers, such as Vistapro, available for home computers. If you'd like to try your hand at ray-tracing height fields, you can buy MetaCreations' Bryce, Animatek World Builder, or World Tool Set. Or you can pick up Craig Kolb's public domain Rayshade ray tracer, which features a very nice implementation of hierarchical grid tracing for height fields. The hierarchical approach captures the best aspects of grid tracing (i.e., low memory overhead) and of quadtree methods (i.e., speed). For multiple renderings of a static height field—as in fly-by animations—the PPT algorithm is the fastest rendering method (Paglieroni 1994).

There are several common file formats for height field data. There is the DEM (digital elevation map) format of the U.S. Geological Survey (USGS) height fields, which contain measured elevation data corresponding to the "quad" topographic maps available from the USGS, which cover the entire United States. The U.S. military establishment has their DTED (digital terrain elevation data) files, which are similar, but are likely to include terrains outside of the United States and its territories. While you may render such data as readily as synthetic fractal terrains, as a synthesist (if you will), I consider the use of "real" data to be cheating! My goal is to synthesize a detailed and familiar-looking reality, entirely from scratch. Therefore, I have rarely concerned myself with measured data sets; I have mostly worked with terrains that I have synthesized myself.

As I have usually worked alone, with no programming assistance, I generally prefer to implement things in the quickest, simplest manner I can readily devise so that I can get on to making pictures. Thus my home-grown height field file format (which is also used by Rayshade) is very simple: it is a binary file containing first a single integer (4 bytes), which specifies the size of the (square) height field, followed by n^2 floats (4 bytes each), where n is the value of the leading integer. I append any additional data I wish to store, such as the minimum and maximum values of the height field, and perhaps the random number generator seed used to generate it, after the elevation data. While far more compact than an ASCII format for the same data, this is still not a particularly efficient storage scheme. Matt Pharr, of ExLuna, has implemented an improved file format, along with conversion routines from my old format to his newer one. In the new scheme, there is a 600-byte header block for comments and documentation of the height field. The elevation data is stored as shorts (2 bytes), with the values normalized and quantized into integers in the range

Team LRN

$[0, 2^{16} - 1]$. The minimum and maximum altitudes over the height field are also stored, so that the altitude values may be restored to their floating-point values at rendering time by the transformation

$$z = \frac{a(z_{max} - z_{min})}{2^{16} - 1} + z_{min}$$

where a is the quantized and scaled altitude value, z is the decoded floating-point value, and z_{min} and z_{max} are the min/max values of the height field. Pharr's code also obviates the big-endian/little-endian byte-order problem that can crop up when transferring binary files between different computers, as well as automatically taking care of transfers between 32-bit and 64-bit architectures. Pharr's code is available on the Internet via anonymous ftp at *cs.princeton.edu*. If you intend to render many height fields, it is worth picking up, as it saves about half of the space required to store a given height field.

HOMOGENEOUS fBm TERRAIN MODELS

The origin of fractal mountains in computer graphics is this: Mandelbrot was working with fBm in one dimension (or one-point-something dimensions, if you must), like the plot we saw in Figure 14.2. He noted that, at a fractal dimension of about 1.2 (the second trace from the top in Figure 14.2), the trace of this function resembled the skyline of a jagged mountain range. In the true spirit of ontogenetic modeling, he reasoned that, if this function were extended to two dimensions, the resulting surface should resemble mountains. Indeed it did, and thus were born fractal mountains for computer graphics. Figure 16.4 is a crude example of such a simple fractal terrain model.

Again, there is no known causal relationship between the shape of real mountains and the shape of this fractal function; the function simply resembles mountains, and does so rather closely. Of course there are many features in real mountains, such as drainage networks and other erosion features, that are not present in this first model. Much of my own research has been toward including such features in synthetic terrain models, largely through procedural methods (Musgrave 1993).

Fractal Dimension

As pointed out in Chapter 14 and as illustrated by Figure 14.2, fractal dimension can be thought of as a measure of the *roughness* of a surface. The higher the fractal dimension, the rougher the surface. Figure 16.5 illustrates how the roughness of an

Team LRN

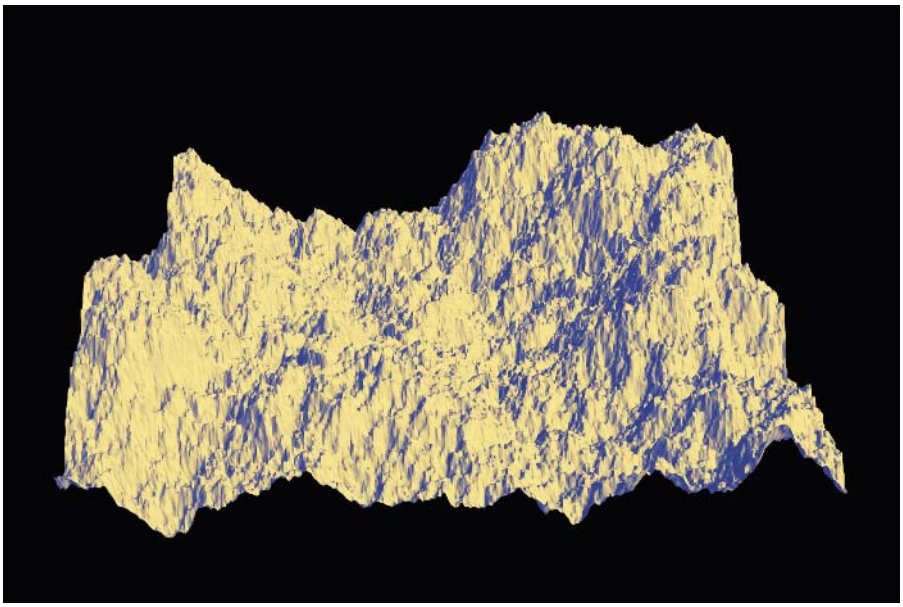


FIGURE 16.4 A terrain patch with homogeneous fractal dimension (of ~2.2).

fBm surface varies with fractal dimension: at the left edge of the patch, the fractal dimension is 2.0; on the right it is 3.0. The most interesting thing about this patch is that it is not planar (i.e., flat) on the left, nor does it fill all of the space on the right. So we see that the formal definition of fractal dimension for fBm does not capture all of the useful fractal behavior available from the construction: the kind of rolling foothills that would occur off the left end of this patch are indeed self-similar and thus fit our heuristic definition of "fractal." Yet they do not fit the formal mathematical definition of fractal dimension (at least not the one for fBm).[3] This is a good example of how fractals defy precise definition and sometimes require that we "paint with a broad brush" so that we don't unnecessarily exclude relevant phenomena. Many researchers in computer graphics and other fields have substituted terms such as "stochastic" and "self-similar" for "fractal" because of this poor fit with formal definitions, but this is probably not appropriate: there are few useful stochastic

3. It's worth noting that different methods for measuring fractal dimension may give slightly different results when applied to the same fractal. So even formal methods may not agree about the limits of fractal behavior and the exact values of quantitative measurements of fractal behavior.

Team LRN

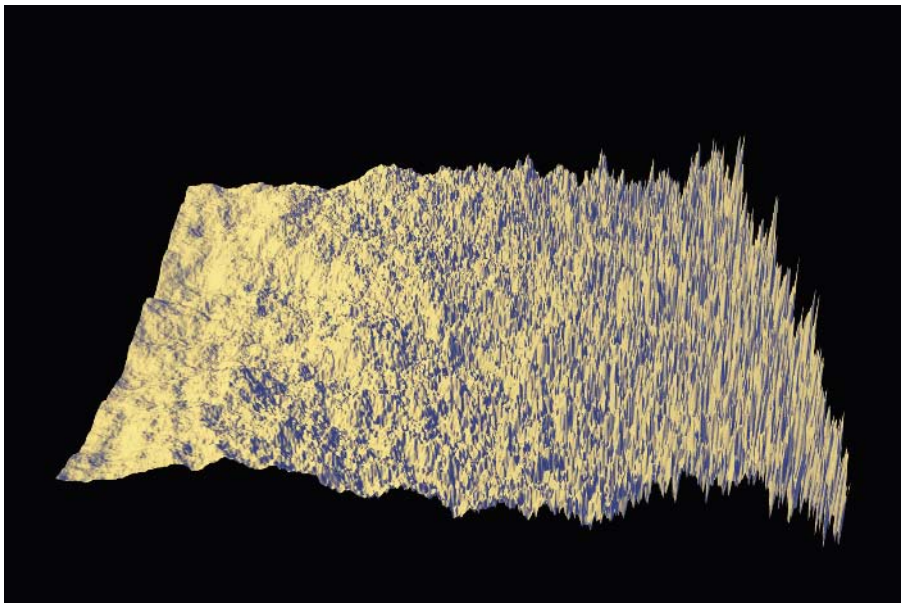


FIGURE 16.5 In this patch, the fractal dimension varies from 2.0 on the left to 3.0 on the right. Copyright © 1994 F. Kenton Musgrave.

models of visual natural phenomena that do not feature self-similarity, and self-similar models are best characterized as fractal, formal technicalities notwithstanding.

Visual Effects of the Basis Function

As illustrated in the previous chapter, the small spectral sums used to create random fractals for computer graphics allow the character of the basis function to show through clearly in the result. Usually, the choice of basis function is implicit in the algorithm: it is a sine wave for Fourier synthesis, a sawtooth wave in polygon subdivision, and a piecewise-cubic Hermite spline in noise-based procedural fBm. You could use a Walsh transform and get square waves as your basis. Wavelets (Ruskai 1992) promise to provide a powerful new set of finite basis functions. And again, sparse convolution (Lewis 1989) or fractal sum of pulses (Lovejoy and Mandelbrot 1985) offer perhaps the greatest flexibility in choice of basis functions. With those methods, you could even use the profile of the kitchen sink as a basis function, leading naturally to sinkholes in the terrain.

Team LRN

Gavin Miller (1986) showed that the creases in terrain constructed with the most common form of polygon subdivision (i.e., subdivision of an equilateral triangle) are really an artifact of the interpolation scheme implicit in the subdivision algorithm. But I think that for the most part it has simply been overlooked that there is a basis function implicit in *any* fBm construction and that the character of that basis function shows through in the result. As shown in the previous chapter, we can use this awareness to obtain certain aesthetic effects when designing both textures and terrains.

The smoothness of the Hermite spline interpolant in the noise function allows us to generate terrains that are more rounded than those commonly seen in computer graphics previously. Figures 15.7, 16.2, and 20.9 illustrate this well. Other examples of basis function effects are seen in Figures 15.15, 16.6, 18.3, 20.18, and 20.20 and in Figures 17.4–17.6, where the ridged basis function was used to get a terrain with razorback ridges at all scales. Note that this terrain model can only be effectively rendered with adaptive level of detail, as with QAEB and other schemes (Bouville 1985; Kajiya 1983b). Without this, in a polygonal model rendered with perspective projection, nearby ridges would take on a saw-toothed appearance, as undersampled elevation values would generally lie on alternating sides of the ridgeline, and distant areas would alias on the screen due to undersampling of the complex terrain model. A nonpolygonal approach to rendering with adaptive level of detail, QAEB tracing, is described in the next chapter. It is ideal for rendering such sharp-edged terrain models.

HETEROGENEOUS TERRAIN MODELS

It would seem that before our 1989 SIGGRAPH paper (Musgrave, Kolb, and Mace 1989) it hadn't yet occurred to anyone to generate heterogeneous terrain models. Earlier published models had been monofractal, that is, composed of some form of fBm with a uniform fractal dimension. Even Voss's heterogeneous terrains (Voss 1988) represent simple exponential vertical scalings of a surface of uniform fractal dimension. As pointed out in Chapter 14, nature is decidedly not so simple and well behaved. Real landscapes are quite heterogeneous, particularly over large scales (e.g., kilometers). Except perhaps on islands, mountains rise out of smoother terrains—witness the dramatic rise of the Rocky Mountains from the relatively flat plains just to their east. Tall ranges like the Rockies, Sierras, and Alps typically have rolling foothills formed largely by the massive earthmovers known as glaciers. All natural terrains, except perhaps recent volcanic ones, bear the scars of erosion. In fact, erosion and tectonics are responsible for nearly all geomorphological features

Team LRN

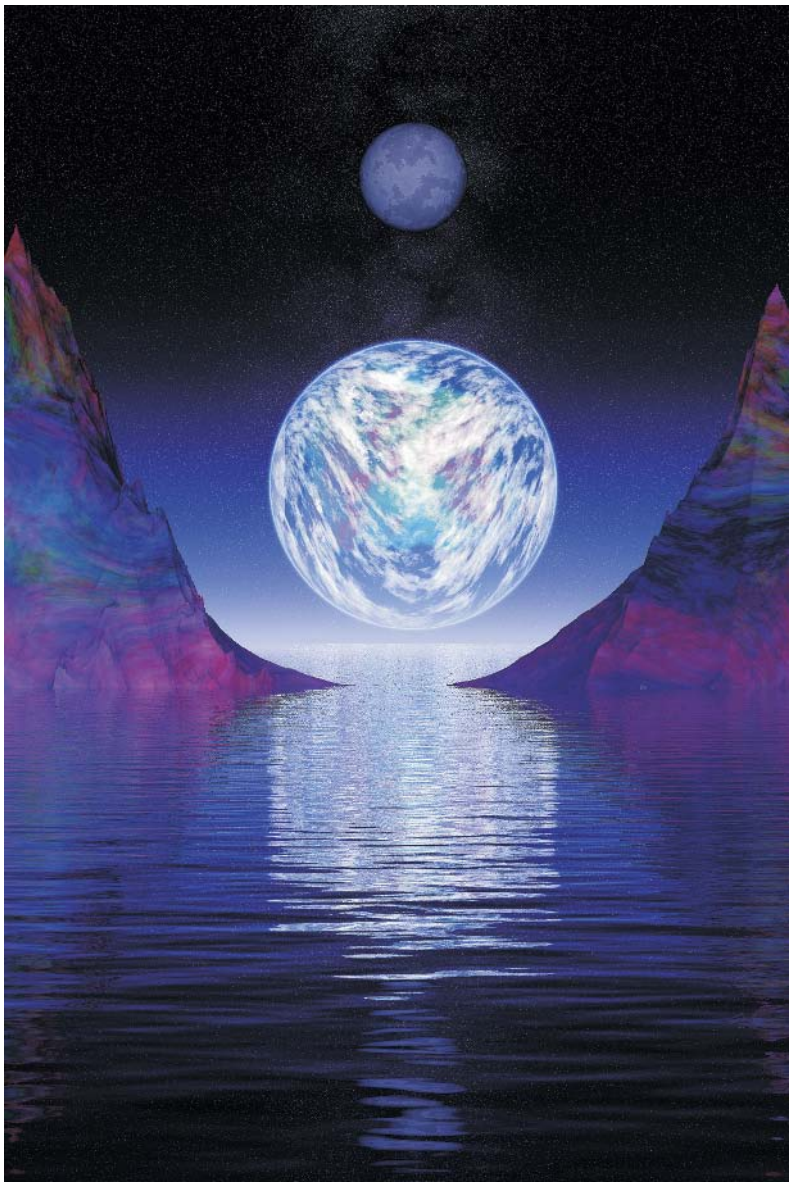


FIGURE 16.6 *Parabolic Curves in the Plane of the Ecliptic* employs most of the tricks described in Chapters 14–18, from multifractals to GIT textures to QAEB tracing. There are parabolas in the terrain, in the central valley, in the clouds, and in depth. Copyright © F. Kenton Musgrave.

Team LRN

on our planet, other than volcanic features, impact craters, and various features due to bioturbation (humanity's included). Some erosion features are relatively easy to model: talus slopes, for example. Others, such as drainage networks, are not so easy (Musgrave, Kolb, and Mace 1989). The rest of this chapter will describe certain ontogenetic models designed to yield a first approximation of certain erosion features, without overly compromising the elegance and computational efficiency of the original fBm model. These models are, at least loosely speaking, varieties of multifractals.

Statistics by Altitude

The observation that motivated my first multifractal model is that, in real terrains, low-lying areas sometimes tend to fill up with silt and become topographically smoother, while erosive processes may tend to keep higher areas more jagged. This can be accomplished with the following variation on fBm:

```
/*
 * Heterogeneous procedural terrain function: stats by altitude method.
 * Evaluated at "point"; returns value stored in "value".
 *
 * Parameters:
 *     "H" determines the fractal dimension of the roughest areas
 *     "lacunarity" is the gap between successive frequencies
 *     "octaves" is the number of frequencies in the fBm
 *     "offset" raises the terrain from "sea level"
 */
double
Hetero_Terrain( Vector point,
         double H, double lacunarity, double octaves, double offset )
{
    double      value, increment, frequency, remainder, Noise3();
    int         i;
    static int first = TRUE;
    static double *exponent_array;

    /* precompute and store spectral weights, for efficiency */
    if ( first ) {
        /* seize required memory for exponent_array */
        exponent_array =
        (double *)malloc( (octaves+1) * sizeof(double) );
        frequency = 1.0;
        for (i=0; i<=octaves; i++) {
             /* compute weight for each frequency */
             exponent_array[i] = pow( frequency, -H );
             frequency *= lacunarity;
         }
         first = FALSE;
```

```
    }

    /* first unscaled octave of function; later octaves are scaled */
    value = offset + Noise3( point ); point.x *= lacunarity;
    point.y *= lacunarity; point.z *= lacunarity;

    /* spectral construction inner loop, where the fractal is built */
    for (1=1; i<octaves; 1++) {
         /* obtain displaced noise value */
         increment = Noise3( point ) + offset;

         /* scale amplitude appropriately for this frequency */
         increment *= exponent_array[i];

         /* scale increment by current "altitude" of function */
         increment *= value;

         /* add increment to "value" */
         value += increment;

         /* raise spatial frequency */
         point.x *= lacunarity;
         point.y *= lacunarity;
         point.z *= lacunarity;
    } /* for */

    /* take care of remainder in "octaves" */
    remainder = octaves - (int)octaves;
    if ( remainder ) {
       /* "i" and spatial freq. are preset in loop above */
       /* note that the main loop code is made shorter here */
       /* you may want to make that loop more like this */
       increment = (Noise3( point ) + offset) * exponent_array[i];
       value += remainder * increment * value;
    }

    return( value );

} /* Hetero_Terrain() */
```

We accomplish our end by multiplying each successive octave by the current value of the function. Thus in areas near zero elevation, or "sea level," higher frequencies will be heavily damped, and the terrain will remain smooth. Higher elevations will not be so damped and will grow jagged as the iteration progresses. Note that we may need to clamp the highest value of the weighting variable to 1.0, to prevent the sum from diverging as we add in more values.

The behavior of this function is illustrated in the terrains seen in Figures 16.2, 16.7, and 18.2.

Team LRN

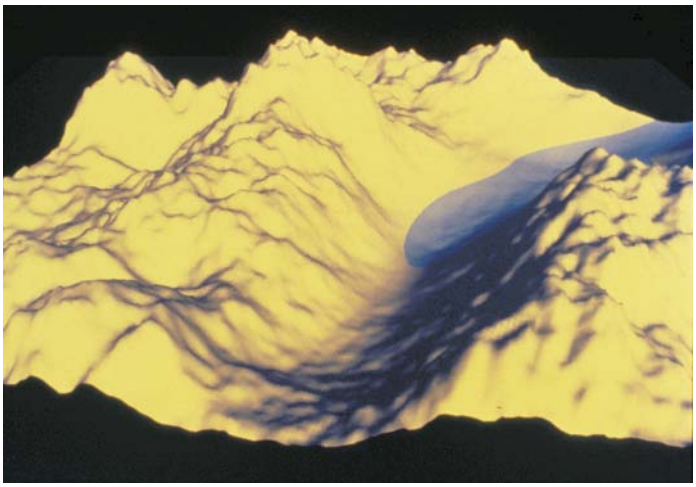


FIGURE 16.7 This multifractal terrain patch is quite smooth at "sea level" and gets rougher as altitude increases. Copyright © 1994 F. Kenton Musgrave.

A Hybrid Multifractal

My next observation was that valleys should have smooth bottoms at all altitudes, not just at sea level. It occurred to me that this could be accomplished by scaling higher frequencies in the summation by the local value of the previous frequency:

```
/* Hybrid additive/multiplicative multifractal terrain model. *
 * Some good parameter values to start with:
 *
 * H:          0.25
 * offset: 0.7
 */
double
HybridMultifractal( Vector point, double H, double lacunarity,
                    double octaves, double offset )
{
    double     frequency, result, signal, weight, remainder;
    double     Noise3();
    int  i;
    static     int first = TRUE;
    static     double *exponent_array;

    /* precompute and store spectral weights */
    if ( first ) {
        /* seize required memory for exponent_array */
        exponent_array =
          (double *)malloc( (octaves+1) * sizeof(double) );
```

```
        frequency = 1.0;
        for (i=0; i<=octaves; i++) {
        /* compute weight for each frequency */
        exponent_array[i] = pow( frequency, -H);
        frequency *= lacunarity;
    }
    first = FALSE;
}

/* get first octave of function */
result = ( Noise3( point ) + offset ) * exponent_array[0];
weight = result;

/* increase frequency */
point.x *= lacunarity;
point.y *= lacunarity;
point.z *= lacunarity;

/* spectral construction inner loop, where the fractal is built */
for (1=1; i<octaves; 1++) {
     /* prevent divergence */
     if ( weight > 1.0 ) weight = 1.0;

     /* get next higher frequency */
     signal = ( Noise3( point ) + offset ) * exponent_array[i];

     /* add it in, weighted by previous freq's local value */
     result += weight * signal;

     /* update the (monotonically decreasing) weighting value */
     /* (this is why H must specify a high fractal dimension) */
     weight *= signal;

     /* increase frequency */
     point.x *= lacunarity;
     point.y *= lacunarity;
     point.z *= lacunarity;
} /* for */

     /* take care of remainder in "octaves" */
     remainder = octaves - (int)octaves;
     if ( remainder )
         /* "i" and spatial freq. are preset in loop above */
         result += remainder * Noise3( point ) * exponent_array[i];

    return( result );

} /* HybridMultifractal() */
```

Note the offset applied to the noise function to move its range from [−1, 1] to something closer to [0, 2]. (If your noise function has a different range, you'll need to

Team LRN

adjust this.) You should experiment with the values of these parameters and observe their effects.

An amusing facet of this function is that it *doesn't* do what I designed it to do: a valley above sea level in this function is defined not by the local value of the last frequency in the sum, as I have assumed, but by the local gradient of the function (i.e., the local tangent, the partial derivatives in x and y—however you care to view it). Put another way, in the above construction, we ignore the bias introduced by lower frequencies—we may be adding a "valley" onto the side of an already steep slope, and thus we may not get a valley at all, only a depression on the side of the slope. Nevertheless, this construction has provided some very nice, heterogeneous terrain models. Figure 16.1 illustrates a terrain model produced from the above function. Note that it begins to capture some of the large-scale heterogeneity of real terrains: we have plains, foothills, and alpine mountains, all in one patch. Figure 20.18 shows a similar construction, this time using the same ridged basis function seen in Figure 20.20: it's like Perlin's original "turbulence" function, which used the absolute value of the noise function, only it's turned upside-down, as 1 − abs(noise) so that the resulting creases stick up as ridges. The resulting multifractal terrain model is illustrated in Figures 17.4–17.6. It is generated by the following code:

```
/* Ridged multifractal terrain model.
 *
 * Some good parameter values to start with:
 *
 * H:          1.0
 * offset: 1.0
 * gain: 2.0
 */
double RidgedMultifractal( Vector point, double H, double lacunarity,
                           double octaves, double offset, double gain )
{
    double     result, frequency, signal, weight, Noise3();
    int        i;
    static int first = TRUE;
    static double *exponent_array;

    /* precompute and store spectral weights */
    if ( first ) {
         /* seize required memory for exponent_array */
         exponent_array =
                 (double *)malloc( (octaves+1) * sizeof(double) );
                 frequency = 1.0;
         for (i=0; i<=octaves; i++) {
         /* compute weight for each frequency */
         exponent_array[i] = pow( frequency, -H );
         frequency *= lacunarity;
```

```
        }
        first = FALSE;
    }

    /* get first octave */
    signal = Noise3( point );

    /* get absolute value of signal (this creates the ridges) */
    if ( signal < 0.0 ) signal = -signal;
    /* invert and translate (note that "offset" should be ~ = 1.0) */
    signal = offset - signal;
    /* square the signal, to increase "sharpness" of ridges */
    signal *= signal;
    /* assign initial values */
    result = signal;
    weight = 1.0;

    for( 1=1; i<octaves; i++ ) {
        /* increase the frequency */
        point.x *= lacunarity;
        point.y *= lacunarity;
        point.z *= lacunarity;

        /* weight successive contributions by previous signal */
        weight = signal * gain;
        if ( weight > 1.0 ) weight = 1.0;
        if ( weight < 0.0 ) weight = 0.0;
        signal = Noise3( point );
        if ( signal < 0.0 ) signal = -signal;
        signal = offset - signal;
        signal *= signal;

        /* weight the contribution */
        signal *= weight;
        result += signal *exponent_array[i];

}

    return( result );

} /* RidgedMultifractal() */
```

Multiplicative Multifractal Terrains

In Chapter 14, Figure 14.3 illustrates, as a terrain patch, the multiplicative multifractal function presented in that chapter. Qualitatively, that terrain patch appears quite similar to the statistics-by-altitude patch seen in Figure 16.1. At the time of this writing, our formal—that is, mathematical, rather than artistic—research into the mathematics of such multifractal terrain models is quite preliminary, so I have little of use to report. The multifractal construction of Chapter 14 does appear to have

Team LRN

some curious properties: as the value of `scale` goes from zero to infinity, the function goes from highly heterogeneous (at zero) to flat (diverging to infinity). We have not yet completed our quantitative study of the behavior, so I cannot elucidate further at this time.

For the time being, however, for the purposes of terrain synthesis it seems best to stick with the two hybrid additive multiplicative multifractal constructions presented in this chapter, rather than attempting to use the pure multifractal function presented in Chapter 14. These hybrid models may be no better understood mathematically, but they *are* better behaved as functions; that is, they don't usually need to be rescaled and are less prone to divergence.

CONCLUSION

I hope that in the last three chapters I have been able to illustrate the power of fractal geometry as a visual language of nature. We have seen that fractals can readily provide nice visual models of fire, earth, air, and water. I hope that I have also helped clarify the bounds of usefulness of fractal models for computer graphics: while fractals are not the final word in describing the world we live in, they do provide an elegant source of visual complexity for synthetic imagery. The accuracy of fractal models of natural phenomena is of an ontogenetic, rather than physical, character: they reflect morphology fairly well, but this semblance does not issue from first principles of physical law, so far as we know. The world we inhabit is more visually complex than we can hope to reproduce in synthetic imagery in the near future, but the simple, inherently procedural complexity of fractals marks a first significant step toward accomplishing such reproduction. I hope that some of the constructions presented here will be useful to you, whether in your own attempts to create synthetic worlds or in more abstract artistic endeavors.

There is plenty of work left to be done in developing fractal models of natural phenomena. Turbulence has yet to be efficiently procedurally modeled to everyone's satisfaction, and multifractals need to be understood and applied in computer graphics. Trees are distinctly fractal, yet to a large extent they still defy our ability to capture their full complexity in a simple, *efficient* model; the same goes for river systems and dielectric breakdown (e.g., lightning). Reproducing other, nonfractal manifestations of heterogeneous complexity will, no doubt, keep image synthesists busy for a long time to come. I like to think that our best synthetic images reflect directly something of our depth—and lack—of understanding of the world we live in. As beautiful and convincing as some of the images may be, they are only a first approximation of the true complexity of nature.

Team LRN

17

QAEB RENDERING FOR PROCEDURAL MODELS

F. KENTON MUSGRAVE

INTRODUCTION

This chapter and the next present some pretty technical discussions. In the previous three chapters, I have tried to keep the discussion at a level where the technically minded artist might want to follow along. Now we're plunging into stuff that is probably only of interest to mathematically minded programmers. So a word of warning: You might just want to read the introductions, which are written at a fairly conversational level, or maybe even skip this chapter and the next entirely. These chapters were written originally as technical papers, so most of the verbiage is in the dry and dense idiom of scientific writing.

In the previous chapter we developed some procedural functions designed to serve as realistic terrain models. When interpreted as height fields, these terrain models are pretty good. But, as we pointed out, height fields are usually precomputed and stored at a fixed *post spacing*. That is, the function is sampled at points on a regular grid, as at the points where lines intersect on graph paper. This has three undesirable side effects: First, we have to store the height field data in files that can become rather large (although this is less an issue as memory and disk space become ever cheaper). Second, the discrete values in the height field must be interpolated, usually linearly, leading to unnatural artifacts such as a terrain composed of triangles—try finding *that* in nature! Third, and the most serious problem in my view, we have a fixed level of detail given by the post spacing. If we get too close to our terrain model, it becomes locally flat and boring. If we view it at too great a distance, we will get aliasing, due to the triangles being smaller than the Nyquist limit, or we have to resort to adaptive level of detail (LOD) schemes.

As demonstrated in earlier chapters, we can build terrain functions that are both continuous (i.e., with a well-defined value everywhere across the surface, not just at certain predetermined sample points) and band-limited so that the features in the model can be kept at or near the Nyquist limit, whatever that limit may be locally.

Team LRN

Thus we can generate models that have ideal appearance everywhere, even though such a model must be view dependent. This is because the appropriate level of detail at any given point is a function of its distance from the view point, due to the perspective projection, as well as the synthetic camera's field of view and resolution. What we require is a rendering algorithm that can take advantage of the flexibility of the procedural approach.

A few years ago I was teaching a class for which the first edition of this book was the text. To illustrate to the class just how simply the procedural approach can generate piles and piles of visual detail, I designed such an algorithm. Again, the goal was maximal simplicity in the algorithm, period. Accordingly, I expected it to be *really* slow. It came as a considerable surprise when it turned out to be only *very* slow, not glacial. (That is, it took on the order of a minute to create an image, when I was expecting days.) I gave this algorithm the wonderfully turgid name *quasi-analytic error-bounded ray tracing,* or *QAEB tracing* for short. To balance the scales of pretense, I pronounce the acronym QAEB whimsically "kweeb" (to rhyme with "dweeb," of course).

QAEB tracing was originally applied to height fields. A discussion with John Hart led me to the realization that, aside from the speedup scheme described later that applies only to height fields, QAEB tracing is actually a general rendering scheme for point-evaluated *implicit* models. Implicit models are isosurfaces of fields, for example, surfaces where some function F defined over three-space is equal to zero. (We'll see examples of exactly this in our cloud models toward the end of this chapter.) So QAEB tracing is actually a pretty powerful rendering method—slow but *really* simple. I should point out that QAEB tracing is simply raymarching but with a variable step size and an implied use of band-limiting in the procedural model to provide adaptive level of detail. But the pertinent point is this: it's simple to implement and it makes cool pictures.

Without further ado, let's launch into the pithy text of the technical paper on QAEB tracing, replete with the turgid use of the royal "we."

QAEB TRACING

We present a numerical method called *QAEB tracing* for ray-tracing procedural height field functions as displacement maps. The method accommodates continuous adaptive level of detail in the height field. Spatial error in ray-surface intersections is bounded by an error specified in screen space. We describe a speedup scheme general to incremental height field ray-tracing methods and a method for rendering

Team LRN

hypertextures, for example, clouds. The QAEB algorithm is simple and surprisingly fast.

One capability distinguishing scanline rendering from ray tracing is the capability of rendering *displacement maps* (Cook 1984). We describe a method for ray tracing a subclass of displacement maps, *height fields*. Height field rendering is important for visualizing terrain data sets, as in various defense-related simulation applications. Adaptive level of detail is desirable in such renderings, and this new method accommodates that simply. Quasi-analytic error-bounded ray tracing is a method for rendering general implicit functions, that is, continuous functions of n variables $F : R^n \rightarrow R$. We will show applications for height fields, where $n = 2$, and hypertextures, where $n = 3$. We call it "quasi-analytic" because it yields a numerical approximation of an analytic ray-surface intersection. This approximation is bounded by an error specified in screen space. When F is a procedural fractal function, the world space frequency content of the terrain model may be linked to its projected screen space Nyquist limit, to accommodate adaptive level of detail without aliasing. QAEB-traced images can be superior, due to their adaptive level of detail and the nonpolygonal character of the rendered height field primitive. As the QAEB algorithm is isolated in the ray-surface intersection routine, QAEB-traced objects amount to new primitives that may be added to the standard inventory of ray-traced geometric primitives such as spheres and polygons. Development of the QAEB algorithm was originally motivated by the desire to render landscapes with adaptive level of detail. We thus describe the algorithm in the context of rendering height fields.

A striking aspect of the QAEB algorithm is its simplicity. It was expected to be slow, due to its profligate character. That is, it requires a very large number of evaluations of the implicit function. In practice it is surprisingly fast, probably due to its simplicity: the entire code can fit in cache on a contemporary microprocessor, yielding near-optimal performance. A desirable feature is that spatial precision (and error) is linked directly to the spatial sampling rate. No greater precision is calculated than is needed, potentially saving CPU cycles and time.

PROBLEM STATEMENT

Generally, the only moving part in a terrain animation is the camera. Freedom of movement of the camera with perspective projection requires adaptive LOD in the terrain model. LOD can be readily accommodated with procedural fractal terrain models. Fractals are scaling (Mandelbrot 1982), thus detail is potentially unlimited.

Team LRN

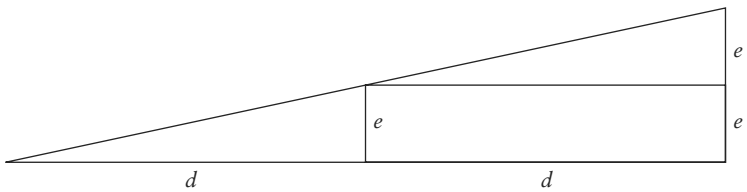


FIGURE 17.1 Feature span e varies linearly with distance d: at base length d, altitude is e; at base length $2d$, altitude is $2e$. For isosceles triangles, reflect about the base.

Procedural fractals are inherently band-limited and can have parameterized band limits, as shown in previous chapters. An argument based on similar triangles (see Figure 17.1) shows that, in a perspective projection, feature span (not area) in screen space varies linearly with distance. We can use this knowledge to roll off high frequencies in the model to keep them at or below the Nyquist limit of the projected basis function (e.g., the Perlin *noise* function) in screen space. Octaves in the spectral construction of the fractal may be related to distance d as

$$octaves = -\log_2(\tan(fov/res)) - \log_2(d) - 3.5$$

where fov is the lateral field of view, res is the number of samples across that field, and 3.5 is a constant relating our chosen Perlin *noise* basis function to the Nyquist frequency.

PRIOR ART

Procedural ray tracing of synthetic fractal terrains with adaptive level of detail has been addressed by Kajiya (1983a, 1983b) and Bouville (1985). These methods are based on polygon subdivision fractal terrains. They require data structure overhead, for example, quadtrees, to organize the numerous polygon primitives. Polygon-based methods can suffer artifacts due to discontinuities in levels of detail. This gives rise to the nontrivial problem of detecting and sealing "cracks" between polygons of different sizes, where adaptive level of detail dictates a change in local polygon size. The QAEB approach is nonpolygonal and features continuous adaptive level of detail, and thus does not suffer from this complication. Procedural fractal functions accommodating continuous frequency change for such adaptive level of detail are described in the previous chapter.

Team LRN

The speedup scheme for rendering height fields was originally proposed by Anderson (1982) and Coquillart (1984). Perlin described a raymarching scheme (Perlin and Hoffert 1989) for hypertextures without LOD.

THE QAEB ALGORITHM

We now describe the QAEB method in the context of the simplest case, rendering height fields.

QAEB tracing is predicated on the following assumptions:

1. A user-specified error ε in the ray-surface intersection and surface normal is acceptable.
2. The function F being rendered is a height field, that is, a continuous function $F : R^2 \rightarrow R$ with a well-defined, globally invariant "up" direction $\vec{u}_F$.
3. F is a point-evaluated function, for example, a procedural texture, that can be efficiently evaluated at any point (x, y).
4. Near and far clipping planes are acceptable.

The algorithm is this: Starting at the near clipping plane, we march away from the view point in error-sized increments until we either exceed the far clipping plane or cross the height field surface, in which case we return an approximate intersection point and surface normal. The marching increment, or *stride* Δ, is exactly equal to ε in the virtual screen's world space dimensions, at the virtual screen's distance from the view point. The size of Δ varies linearly with distance, as shown in Figure 17.1. At each step we evaluate the height field function F and compare it against the ray's altitude relative to $\vec{u}_F$. (Note that this constitutes rendering an implicit function with the isosurface $F(x,y)$ = surface elevation.) This is profligate in evaluations of F, leading to two conclusions: the algorithm is expected to be slow, and F should be as efficient as possible.

C code implementing the QAEB algorithm is presented in Appendix A.

ERROR IN THE ALGORITHM

The error is defined to be the difference between the analytic intersection of the ray and F, and the intersection determined by the QAEB approximation. The error ε is specified in screen space in terms of sample spacing, for example, one pixel. Both the

Team LRN

stride and the error vary in world space proportional to εd, where d is the distance from the view point. The value of ε actually specifies three somewhat independent errors along three axes: the vertical, lateral, and depth axes of the (horizontal) screen. The lateral error is exactly ε. The vertical error (the error associated with vertical perturbation of the sampling ray) is indeterminate, due to the chances of hitting or missing a ridge profile. Hitting or missing the top of a ridge can result in vastly different intersection points in world space; this problem is intractable. The depth error is more interesting. To ensure that depth error corresponds to ε, it should be proportional to the slope F' of F. For F with F' discontinuous at local maxima and maximum slope $(F_{\max})'_{\max}$ at such maxima, the stride Δ should vary as $1/(F_{\max})'_{\max}$ to ensure meeting the specified error.[1] If F yields low, smooth terrain, Δ may be increased. Note that *fractional Brownian motion* (fBm), upon which most synthetic terrain models are based, is self-affine (Voss 1988). That is, for fBm, local slope can increase as higher spatial frequencies are added. Thus the correct stride may change with the dictates of LOD on terrain frequency content. (Note that these matters notwithstanding, we have never found it necessary to use anything other than the simple stride length εd in our applications for image synthesis of nonpathological models—in other words, anything we ever wanted to render nicely.[2])

NEAR AND FAR CLIPPING PLANES

Stride Δ varies linearly with distance d from the view point. It follows that at distance $d = 0$, $\Delta = 0$. Therefore, raymarching must begin at a near clipping distance $d_0 > 0$. Features closer to the view point than d_0 will not be visible. Conversely, a greater value of d_0 implies a larger initial stride. Thus the value of d_0 can significantly impact rendering time.

---

1. This statement is actually incorrect; the truth is more subtle. The depth error, and thus the stride, should be linked to the minimum slope occurring at local maxima of F. That is, if local maxima may be very narrow (corresponding to sharp peaks or ridges in the terrain), the stride must be small enough to capture them within the specified error. However, if maxima of F may be steep on one side only, the error corresponds to the minimum $(F_{\max})'_{\min}$ of the two slopes on either side of the local maximum, as $1/(F_{\max})'_{\min}$.

2. Manuel Gamito has used interval arithmetic to ensure analytic ray-surface intersections in the QAEB scheme and ray-domain distortion methods (Barr 1986) to render nonheight field terrains as seen in Figure 17.2. Werner Benger (*www.photon.at/ ~werner/Light.html*) has developed more approximate marching methods to speed up the QAEB rendering process, while sacrificing accuracy in the ray-surface intersection.

Team LRN

FIGURE 17.2 A QAEB-traced non-height-field model of a wave, modeled and rendered by Manuel Gamito. The spray and foam are QAEB hypertextures.

A far clipping plane[3] $d_f < \infty$ is necessary to terminate computation. Beyond d_f the model is not visible. Smaller values of d_f imply a shorter raymarch. Choosing the distances for the clipping planes involves the kind of trade-off between time and realism typical in computer graphics.

CALCULATING THE INTERSECTION POINT AND SURFACE NORMAL

Ray-surface intersection is indicated when the ray altitude crosses adjacent evaluations z_{i-1} and z_i of F. As these evaluation points are exactly ε apart, either can serve as the approximate intersection point, both values being guaranteed to be within ε. Alternatively, we may use the intersection of the ray and the line between z_{i-1} and z_i. This may yield a slightly more accurate solution at a cost of a few more operations—a cost that will generally be overwhelmed by the cost of the march to the intersection.

3. We refer to the near and far clipping distances as "clipping planes," although they are implemented as distances from the view point and are thus more like "clipping spheres."

Team LRN

The surface normal is constructed via the cross product of two vectors between three samples of F. The first vector $\vec{v}_d$ is from z_{i-1} to z_i. The second vector $\vec{v}_l$ is from z_{i-1} to a third sample z_l of F taken at a distance equal to the current stride Δ_i in a lateral direction, that is, perpendicular to the plane containing the ray and $\vec{u}_F$. The normalized cross product $\|\vec{v}_d \times \vec{v}_l\|$ serves as our surface normal approximation.

C code implementing this scheme appears in Appendix B.

ANTIALIASING

Antialiasing may be accomplished with ordinary supersampling methods. Supersampling implies sampling at a higher spatial frequency; this in turn implies a smaller error ε. In adaptive supersampling, this may require that the samples that indicate supersampling be recomputed with the implied smaller ε to ensure meaningful results. Uniform supersampling simply implies a smaller ε throughout, as ε should generally be equal to the screen space distance between adjacent samples.

To the extent that F represents an uncorrelated random function (all reasonable terrain models being in fact highly correlated), the model is self-jittering. That is, samples automatically have a stochastic character, even if equidistant in screen space. This is a property inherent in the random fractal model, not the QAEB rendering method.

A SPEEDUP SCHEME FOR HEIGHT FIELDS

Assuming that (1) the scene is rendered bottom to top relative to $\vec{u}_F$ and (2) $\vec{u}_F$ coincides with the screen "up" vector $\vec{u}_s$, we may employ a simple optimization to great benefit. We keep an array A_d of depth values of size equal to the number of lateral samples in the screen. A_d is initialized to d_0 and updated to the distance d_i from the view point of the last intersection in the corresponding column. Subsequent rays in the same column, but higher relative to $\vec{u}_F$, may begin marching at d_i rather than d_0. This speeds up rendering enormously, particularly for horizontal views of great depth. The second assumption may be dispensed with by indexing A_d along an axis $\vec{v}_F$ horizontal relative to $\vec{u}_F$ and in the plane of the virtual screen. The span w_d of the array indices along $\vec{v}_F$ is equal to the extent of the projection of the rotated screen onto $\vec{v}_F$ (see Figure 17.3). The number of buckets in A_d is $\lceil w_A/\varepsilon \rceil$. A given screen sample is projected onto $\vec{v}_F$ to determine its bucket in A_d. This bucket is within the specified error ε, laterally. The projection of a point p on the screen onto $\vec{v}_F$ is

Team LRN

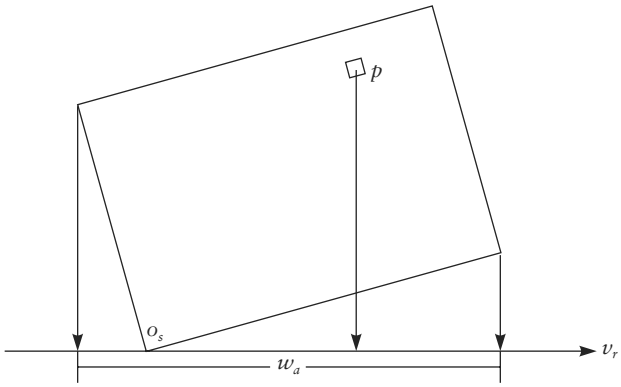


FIGURE 17.3 Projection of rotated screen and sample onto horizontal axis.

$\vec{v}_F\lceil(p - o_S)\bullet\vec{v}_F\rceil$, where o_s is the lower-left corner of the screen relative to $\vec{u}_F$ and $\vec{v}_F$. This speedup scheme is complicated in cases where the screen contains the point where $\vec{u}_F$ and the view direction are parallel. Such cases may be dealt with by keeping two depth arrays, one for each of the two marching directions opposite relative to $\vec{u}_F$.

SHADOWS, REFLECTION, AND REFRACTION

In rendering with adaptive level of detail, correct shadows depend on correct projected shadow feature size. In the QAEB scheme with a given ε, that size is equal to the stride Δ_i at the intersection point where the shadow ray is spawned. Features must retain this size in the shadow projection. For a light source at infinity, the feature size does not vary with the projection. Thus the shadow ray stride is constant and equal to Δ_i. For a light source not at infinity, for example, a point light source, feature size changes linearly with distance, as shown earlier. It follows that shadow rays start with stride proportional to Δ_i, and the stride goes to zero as the distance to the light source d_l goes to zero. Stride then varies with Δ_i and d_l as $\Delta_i d_l/(d_l + 1)$.

Reflection and refraction cannot be handled correctly in the QAEB scheme, as the divergence of adjacent rays is affected arbitrarily in such (specular) transport, and our assumptions about the geometric error break down. This may not be significant, as this scheme was developed to render fractal models of natural phenomena, such as landscapes and clouds, that usually feature neither type of specular

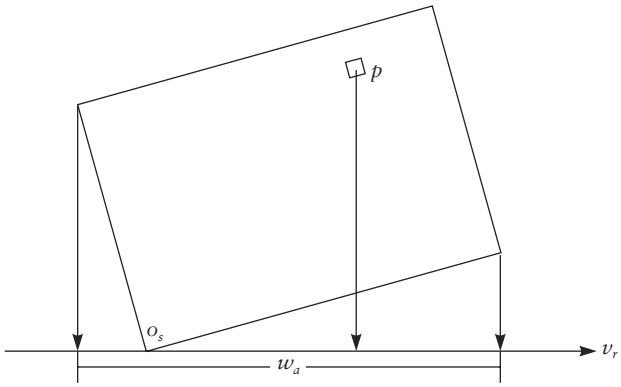


FIGURE 17.3 Projection of rotated screen and sample onto horizontal axis.

$\vec{v}_F\lceil(p - o_S)\bullet\vec{v}_F\rceil$, where o_s is the lower-left corner of the screen relative to $\vec{u}_F$ and $\vec{v}_F$. This speedup scheme is complicated in cases where the screen contains the point where $\vec{u}_F$ and the view direction are parallel. Such cases may be dealt with by keeping two depth arrays, one for each of the two marching directions opposite relative to $\vec{u}_F$.

SHADOWS, REFLECTION, AND REFRACTION

In rendering with adaptive level of detail, correct shadows depend on correct projected shadow feature size. In the QAEB scheme with a given ε, that size is equal to the stride Δ_i at the intersection point where the shadow ray is spawned. Features must retain this size in the shadow projection. For a light source at infinity, the feature size does not vary with the projection. Thus the shadow ray stride is constant and equal to Δ_i. For a light source not at infinity, for example, a point light source, feature size changes linearly with distance, as shown earlier. It follows that shadow rays start with stride proportional to Δ_i, and the stride goes to zero as the distance to the light source d_l goes to zero. Stride then varies with Δ_i and d_l as $\Delta_i d_l/(d_l + 1)$.

Reflection and refraction cannot be handled correctly in the QAEB scheme, as the divergence of adjacent rays is affected arbitrarily in such (specular) transport, and our assumptions about the geometric error break down. This may not be significant, as this scheme was developed to render fractal models of natural phenomena, such as landscapes and clouds, that usually feature neither type of specular

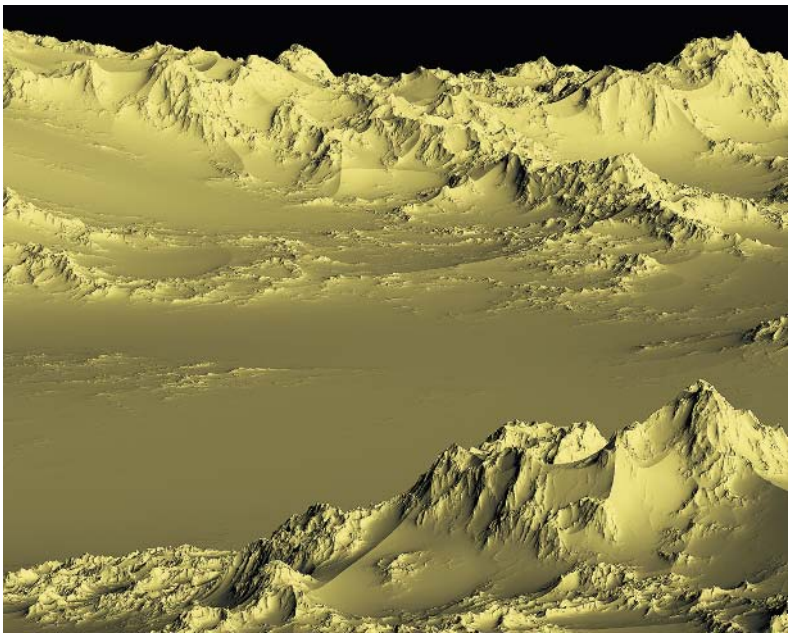


FIGURE 17.4 The very first QAEB-traced terrain. The terrain model is the "ridged multifractal" function described in Chapter 16. Copyright © F. Kenton Musgrave.

transport. Experience indicates that reflective and refractive stochastic surfaces generally create visually confusing images, at any rate.

PERFORMANCE

The QAEB-traced terrain in Figure 17.4 was rendered in 2 minutes, 41 seconds on a 150 MHz R4400, at 640 × 480 (NTSC video) resolution. Near and far clipping planes are at 0.01 and 100.0, respectively. Figure 17.5 is at the same resolution, with shadows from a directional light source (i.e., a light source at infinity) at an elevation of ~30° above horizontal. Rendering time was 17 minutes, 57 seconds. Figure 17.6 was rendered at a film recorder resolution of 2000 × 1333, without shadows and with atmospherics, in 62 minutes. Near and far clipping planes are at 0.4 and 100.0, respectively. All images were rendered at one ray per pixel with an ε of one pixel.

This performance is far better than was expected before testing. Expectations were low because of the profligate evaluations of the procedural height field

Team LRN

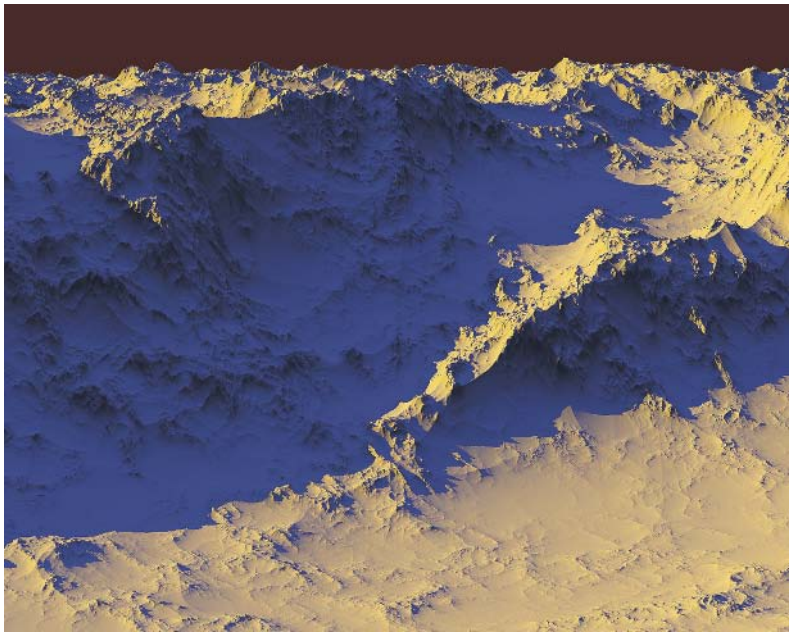


FIGURE 17.5 The first QAEB rendering demonstrating shadows from a light source at infinity. Copyright © F. Kenton Musgrave.

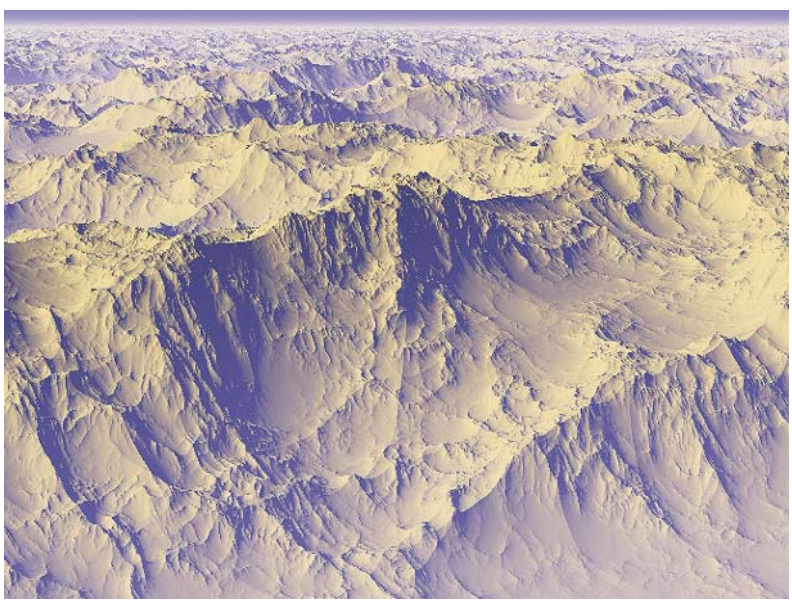


FIGURE 17.6 High resolution and great depth, with atmosphere.

function, which uses on the order of 10^3 floating-point operations per evaluation. QAEB tracing time is dominated by evaluations of the Perlin *noise* function in our implementation. Rendering time is directly impacted by the computational complexity of *F,* the size of the stride, the screen resolution, the number of screen samples, the distances to the near and far clipping planes, the angle of the view direction relative to $\vec{u}_F$, the use of shadows, and, in that case, the number of light sources and their angles to $\vec{u}_F$ and the slope considerations at local maxima described earlier.

QAEB-TRACED HYPERTEXTURES

The QAEB scheme is readily applicable, without the previous speedup scheme, to volume rendering of procedural hypertextures (see Figure 17.7).

FIGURE 17.7 QAEB-traced scene with volumetric shadowing.

Clouds

Rather than checking ray height versus F along the raymarch, you can interpret values of $F > 0$ as density, with values of $F < 0$ being zero density or clear air. This corresponds to raymarching a *hypertexture* (see Chapter 12). Accumulating the density according to Beer's law (see Chapter 18) and computing lighting with a self-occluding, single-scattering model can yield nice results.[4] Applying a realistic high-albedo, anisotropic multiple-scattering illumination model (Max 1994) yields substantially better results, as seen in Figures 17.8 and 17.9.[5]

Our hypertexture cloud model has two parts: a procedural fractal function, usually either vanilla fBm or Perlin's "turbulence" (fBm constructed using the absolute value of Perlin *noise* as the basis), and a simple weighting function that modulates the fractal, making the fractal cloud a more or less distinct blob situated in clear air. The weighting function is necessary to ensure that fractal clouds do not completely permeate all of space, as the fractal functions are statistically homogeneous. This weighting function can be as simple as $F = 1.0 - d$, where d is the distance from a central point, or something more complex to shape the cloud, as seen in Figure 17.9 and Figure 17.8(a), where the cloud bottom is rather flat. (The shaping function for these clouds is included in the C code on this book's Web site, *www.mkp.com/tm3*.)

In the simplest single-scattering illumination model, local illumination is attenuated by accumulating density along a shadow ray shot toward the light source. Naively, one such ray must be sent per sample, accounting for most of the cost in the rendering. In practice, the frequency of such illumination samples can be decreased, and their stride length may be increased, up to the point of objectionable artifacts. (For details on how, see the code on this book's Web site.) Storing precomputed illumination values in a voxel grid can speed rendering at the cost of increased memory use.

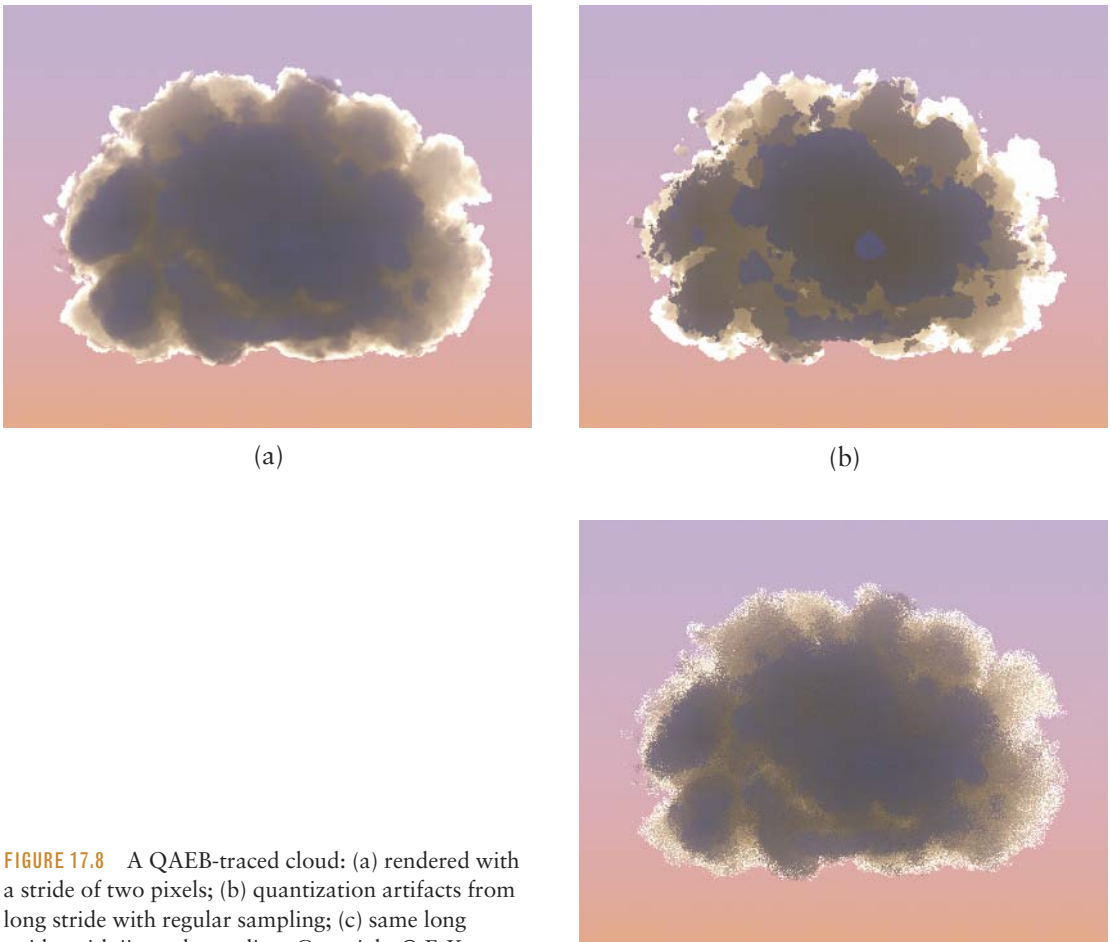
Jittering samples within the interval of the stride (Cook 1984) also allows greater stride length in primary and shadow rays, with graceful degradation in image quality, as seen in Figure 17.8(c). Nonjittered samples will lead to conspicuous quantization artifacts at large stride lengths, as illustrated in Figure 17.8(b).

---

4. Note that in this application, QAEB tracing is a slower but simpler and more accurate equivalent of the gas rendering methods in Chapter 9.

5. Unfortunately, treatment of such physical illumination models are beyond the scope of this book. Their development is the current Ph.D. research topic of my student Sang Yoon Lee at the time of this writing. For more on his work, see *http://student.seas.gwu.edu/~sylee/*.

Team LRN



(a) (b) (c)

FIGURE 17.8 A QAEB-traced cloud: (a) rendered with a stride of two pixels; (b) quantization artifacts from long stride with regular sampling; (c) same long stride, with jittered sampling. Copyright © F. Kenton Musgrvae.

Simple QAEB-traced fBm clouds with single-scattering illumination were used to create the Mickey Mouse and Goofy characters and a rather detailed cruise ship model in a television commercial for Disney cruise lines produced during my tenure at Digital Domain. Only spherical cloud primitives were used, along with animation of the hypertexture spaces and densities. These simple models yielded some striking and novel animation effects. For a preliminary cloud character animation test from this project, see *www.kenmusgrave.com/animations.html*.

Team LRN

FIGURE 17.9. A QAEB-traced hypertexture cloud, with a radiosity solution for high-albedo anisotropic multiple scattering, by Sang Yoon Lee. The radiosity model is by Nelson Max.

BILLOWING CLOUDS, PYROCLASTIC FLOWS, AND FIREBALLS

An ontogenetic model of billowing can yield realistic dynamic models of fireballs, billowing smoke, and growing cumulus clouds. The distinctive behavior we call "billowing" results from rapid advection in a fluid medium, causing turbulent flow in three dimensions. Accurate modeling of such turbulent flow is a problem of notorious computational difficulty, which only recently yielded to solutions practical for the field of image synthesis (Fedkiw 2001). For entertainment applications, empirical accuracy is not the goal; visual verisimilitude is, and visual novelty might prove even more useful.

The film *Dante's Peak* required effects simulating the fast-moving, highly destructive cloud of hot volcanic ash known as a pyroclastic flow. At the time, I had already developed the realistic hypertexture cloud model just described. That model was originally designed to serve as a testbed for the difficult problem of modeling the anisotropic, high-albedo, multiple-scattering illumination required for truly realistic

Team LRN

rendering of clouds, as seen in Figure 17.9. That cloud model can be extended to model billowing in a relatively simple way.

Our first attempt consisted of a swarm of cloudlets animated in the Alias Dynamation particle system package. This approach yielded good results when the animation consisted simply of sweeping the swarm of cloudlet fields through a static texture space. "Popping" of high frequencies is inevitable in such a scheme because the higher-frequency details change faster than those of lower frequency as the cloud front advances through texture space. The result was an explosive quality in the advancing cloud, which seemed appropriate. However, the director, citing footage of real volcanic clouds, requested a dynamic, billowing quality wherein the cloud appears to turn inside out as it evolves along its forward direction. This leads to the solution presented here.

A single cloudlet can be made to appear to billow by scaling the domain of the fractal function, relative to the "forward" direction of the billowing. The scaling is of the angle a given sample makes with that axis with time (see Figure 17.10). The result of this domain distortion is that, as the sample point is rotated toward the forward direction in magnitude proportional to the angle, cloud features appear to rotate toward a singularity opposite the "forward" direction.

Features get stretched longitudinally with time in this scheme. To ameliorate objectionable artifacts arising from this, the distorted, "old" texture is rolled off with time and replaced with a "young," undistorted texture that is in turn rolled off as it ages. This proceeds cyclically, yielding a cyclic model. This cyclic nature is disguised, visually, by growth in size of the cloudlet with time. For details on how this is accomplished, see the code on this book's Web site (*www.mkp.com/tm3*).

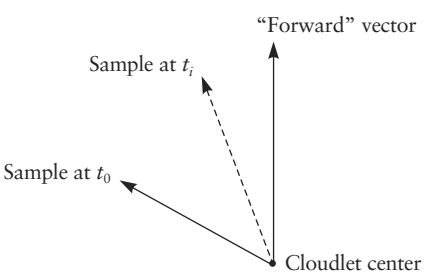


FIGURE 17.10 Rotation of samples with time in the billowing scheme.

Team LRN

Fireballs

Fireballs may be simulated by animating a color map that varies from white through yellow, orange, and red to black, as in the "flame" shader in Chapter 15. This map is indexed by radius (hotter toward the center) and time (cooling to black with age). The map color is used as the cloud texture color. Fireballs are more efficient to render than clouds because they are self-luminous and require no illumination or shadow calculations. Preliminary models of such CG pyrotechnics were developed for a bid for certain special effects scenes in the film *Air Force One*.[6]

Psychedelic Clouds

As noted in Chapter 15, a vector-valued fBm can be interpreted as an RGB color vector and used to color a cloud to get a fantastical coloration, as seen in Figure 17.11. The animation *Comet Leary*[7] was accomplished with this model, by sweeping the hypertexture through five spherical weighting fields of successively decreasing density. Though not a convincing simulation of turbulent flow, it is a simple model yielding a visually appealing effect. The GIT scheme described in Chapter 15 could be used to obtain greater control over stochastic coloring.

CONCLUSION

We have demonstrated a numerical algorithm for ray-tracing implicit procedural functions with adaptive level of detail. It is simple, surprisingly fast, and general to all continuous functions $F : R^n \rightarrow R$. It features a screen space geometric error bounded by a user-specified value. The described speedup scheme, which is important to good performance in rendering height fields, is general to all incremental height field ray-tracing schemes. The stride we employ is the most conservative possible. It may be possible to use more sophisticated methods, such as Lipschitz conditions (Standler and Hart 1994; Worley and Hart 1996) to speed rendering by extending the average stride length. The demonstrated speed of this algorithm is surprising. This speed is conjectured to be linked to the algorithm's simplicity, which allows it to reside entirely in cache memory for near-optimum microprocessor performance.

---

6. For an early animation test, see *www.kenmusgrave.com/animations.html*.
7. Also available at *www.kenmusgrave.com/animations.html*.

Team LRN

FIGURE 17.11 *Comet Leary* is a whimsical rendering of Timothy's final return to Earth. It is a QAEB cloud with coloring by a vector-valued fBm. Copyright © F. Kenton Musgrave.

We have adapted the method to render measured height field data sets to add adaptive level of detail. For this application, we simply constrain the fractal function F by the measured data to render such data sets with added stochastic detail. The lowest frequency in the added fractal detail should be close to that of the post spacing in the measured data set.

We have also extended the method to procedural hypertextures, with a corresponding increase in rendering time. The results have been cloud models of unprecedented realism, some promising synthetic pyrotechnic effects, and some gratuitous psychedelia.

Team LRN

18

ATMOSPHERIC MODELS

F. KENTON MUSGRAVE, LARRY GRITZ,
AND STEVEN WORLEY

INTRODUCTION

Even more than the last chapter, this chapter is highly technical and may be of interest only to mathematically minded programmers. You might just want to read this introduction, which is written at a fairly conversational level. The rest of this chapter was originally written as a technical paper, so most of the prose is pretty dry and terse.

Rendering realistic atmospheric effects involves three distinct elements: scattering models, geometric atmospheric density distribution (GADD) models, and numerical integration schemes for each. We present a series of simple, continuous GADD models of increasing geometric fidelity to spherical planetary atmospheres and integration schemes for each. We describe a RenderMan implementation of a planetary atmosphere and a general GADD integration scheme with bounded numerical error. We also suggest a simplified approximation of Rayleigh scattering. These models are distinctly nonphysical, ontogenetic models designed to be useful for production image synthesis rather than to provide accurate simulations. The GADDs we present and their associated integration schemes can, however, be coupled to published physical scattering models to augment the accuracy of those models.

Landscape painters have known for hundreds of years that *aerial perspective,* the bluing and loss of contrast with distance, is the primary visual cue indicating large physical scale in a rendering (Gedzelman 1991).[1] These effects are due to

1. Shannon (1995) points out that artists break aerial perspective into two components: *atmospheric perspective* and *color perspective.* The former is the change in contrast with distance, due to noncolored haze. The latter is the change in color with distance, light backdrops becoming redder and dark ones bluer, due to Rayleigh scattering. For more on this topic see *www.kenmusgrave.com/4_persp.html.*

Team LRN

atmospheric scattering of light. In the scientific literature these effects are described by the Rayleigh and Mie scattering models. Previous authors (Klassen 1987; Nishita et al. 1993; Tadamura et al. 1993) have presented physical models of Rayleigh and Mie scattering. Efficiency of light scattering is modulated in part by the density of the atmosphere, which in turn varies spatially. This variation is not only with altitude: altitude is a function of the radius from the center of a planet, thus at the largest scales, the atmosphere must curve around a planet. We call a model describing the spatial distribution of atmospheric density *geometric atmospheric density distribution* (GADD). In Klassen (1987) the GADD consists of two horizontal slabs of constant density; Nishita et al. (1993) uses concentric shells of linearly interpolated density. We present some continuous GADDs to improve upon those models.

The global context for a landscape is the surface of a planet. In that context, all optical paths, defined as a ray's extent in a participating medium, are finite since they either intersect a surface, are absorbed or scattered, or exit the atmosphere due to its curvature around the planet. The spatial distribution of a GADD can affect its accuracy as a function of scale: the simplest, homogeneous GADD is accurate for small scales, an intermediate-scale model may take into account change in density with height, while a global GADD should take into account both change in density with altitude and the curvature of the atmosphere. (Modeling the true complexity of atmospheric structure remains impractical.) We will describe both local and global GADD models, along with methods for integrating their optical density along arbitrary ray paths. The local GADD models may be integrated analytically. We will describe numerical integration schemes for the global GADD functions. We also suggest a simple, computationally minimal approximation of Rayleigh scattering. Although nonphysical, this model produces an artistically sufficient model of aerial perspective without the unnecessary complications of physical models.

Although the models presented here are not physical models, they may in some cases represent improvements over published "physical" models. We claim that they are visually effective and that they are recommended by Occam's Razor due to their simplicity. Most are not particularly novel, as they appear to have been repeatedly reinvented by different graphics researchers. Nevertheless, they have yet to be described in the mainstream graphics literature.

Blinn (1982a) pointed out that atmospherics involve two distinct types of models: scattering models and density distribution models. We assert that integration schemes for these models constitute a third essential element. Our goal is to address all three elements, to visual satisfaction, in as little computation as possible.

Team LRN

For our discussion, we define the *optical path* as the extent of a ray's passage through an atmosphere, *scattering* as redirection of direct illumination from a light source (implying single scattering) into the optical path and toward the view point, *outscattering* as redirection of light out of an optical path toward the view point, and *extinction* as the cumulative effect of both outscattering and absorption along the path.

BEER'S LAW AND HOMOGENEOUS FOG

As a light ray traverses an optical path, some light is extinguished and some light may be added by emission and/or scattering. As described in Max (1986), the sum of these effects can describe, physically, the behavior of a participating medium. (We discuss only atmospheres here, not general participating media such as glass, water, smoke, flames, clouds, and so forth.)

The effect of an atmosphere on the intensity of a light ray can be described by the differential equation

$$dI = \sigma(\vec{x}) + E(\vec{x})\,d\vec{x}$$

where x is the position in three dimensions, $\sigma(\vec{x})$ describes extinction per unit length, and $E(\vec{x})$ describes emission and scattering per unit length into the optical path. When σ and E are proportional to one another and are functions solely of position x, we can define *optical depth* τ as

$$\tau = \int_0^{t_e} \sigma(\vec{x})\,dt$$

the integral over the optical path of the GADD $\sigma(\vec{x})$. As in Haines (1989) we index the position along the ray by t, which ranges from 0 to t_e. Beer's law gives a physical solution to this simple model and gives us the transparency T over the optical path as a function of τ:

$$T = e^{-\tau}$$

For a homogeneous and isotropic GADD (i.e., $\sigma = c$, a constant), we have $\tau = ct_e$. Such homogeneous fog is fairly common in renderers; its effectiveness is due to its embodiment of a simple but physically accurate scattering model.

For simplicity, we consider extinction over the optical path to be $1 - T$. This extinguished portion of the intensity is replaced with the atmosphere color as a direct

Team LRN

consequence of the $E(\vec{x})$ term in the previous differential equation. In the models we present below, we ignore emission[2] but engineer the absorption rate σ to vary spatially. After obtaining τ by integrating σ, we may use Beer's law to accurately compute the true effect of the atmosphere, given a scattering model. We therefore concentrate most of the rest of our presentation on nonhomogeneous density models and their integration.

EXPONENTIAL MIST

A useful GADD for landscape renderings has a density distribution that varies as e^{-z}, where z is altitude relative to a horizontal plane. It features local fidelity to nature, as atmospheric optical density is known to vary exponentially with altitude (Lynch 1991). Behavior of this GADD can be parameterized as

$$\sigma = Ae^{-Bz}$$

where A controls the overall density and B controls the falloff of the density with altitude. Its effectiveness in a landscape rendering is illustrated in Figure 18.2. This GADD may be integrated analytically:

$$\tau = t_e \int_{z_o}^{z_o} e^{-z} dz = \frac{At_e}{Bz_d}\left(e^{-(z_o + z_d t_e)} - e^{-z_o}\right)$$

where z_0 is the z coordinate of the ray origin and z_d is the z component of the ray's normalized direction vector. For very small z_d, corresponding to nearly horizontal rays, we substitute

$$\tau = At_e e^{-Bz_o}$$

2. Although we do not model emission, a similar effect is nonetheless obtained. These are nonphysical scattering models, in which conservation of energy is not maintained. Because the atmosphere color does not depend on illumination, except in our last model, energy may appear in the optical path without illumination (e.g., the white color of an atmosphere that is actually in shadow). This nonphysicality can be used to artistic advantage, as illustrated in Figure 18.1, a rendering without any light sources. The color variation in that atmosphere is attained basically by reversing the order of the RGB values for the extinction coefficients on our simplified Rayleigh scattering model, from those used to get blue sky.

Team LRN

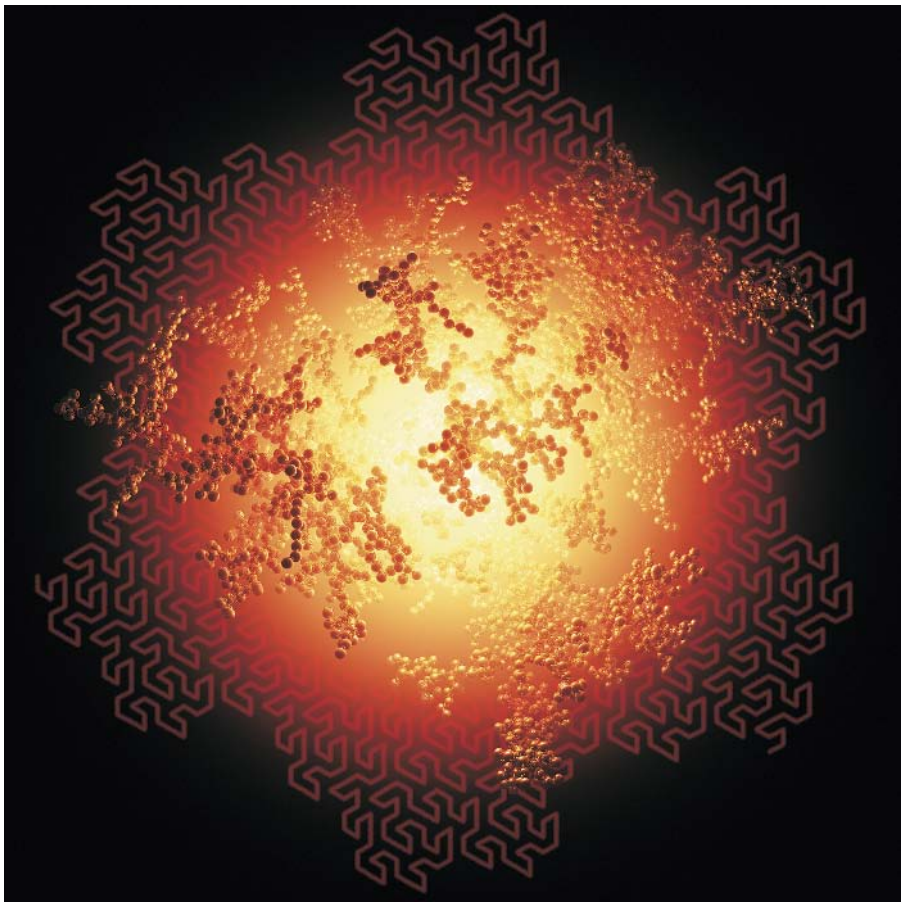


FIGURE 18.1 *Fractal Mandala* illustrates an alternative use of the planetary atmosphere and minimal Rayleigh scattering model. The color is obtained by changing values in the extinction coefficient vector from those used for Rayleigh scattering. Copyright © F. Kenton Musgrave.

Team LRN

FIGURE 18.2 *Carolina* illustrates both the exponential atmosphere and color perspective. Copyright © F. Kenton Musgrave.

the asymptotic value as z_d goes to zero, to prevent division by zero. This GADD has unbounded optical paths for horizontal rays. The optical path of such rays is ultimately limited by the finite numerical representation of infinity in the renderer; this value can be varied to limit integration.

A RADIALLY SYMMETRIC PLANETARY ATMOSPHERE

Because of the infinite optical paths cited earlier, a GADD that varies exponentially with radius from a central point is more realistic on large scales. For such a GADD, the radius r from the central point is related to position t along the ray as

$$r(t) = \sqrt{\alpha^2 + 2\beta t + t^2}$$

where α and β are constants determined by the ray path and the origin, or center, of the radial fog. This function forms a hyperbola. With σ given as this function of r, instead of computing

Team LRN

$$\int \sigma(t)\,dt$$

we must compute

$$\int \sigma\left(\sqrt{\alpha^2 + 2\beta t + t^2}\right)dt$$

which can prove challenging.

A simple radial GADD is

$$\sigma(r) = e^{-r^2}$$

This function cannot be integrated in closed form, but a numerical approximation is available in the C and FORTRAN math libraries in the *error function:*

$$erf(x) \approx \frac{2}{\pi}\int_0^x e^{-t^2}$$

This GADD may be parameterized as

$$\sigma = Ae^{-Br^2}$$

where A controls the overall density and B modulates falloff with radius. This GADD can provide convincing visual results (see Figure 20.4).

Consider integrating this GADD. For a ray origin $\vec{r}_0$ and unit direction $\vec{x}$, and a GADD origin $\vec{o}$, we have

$$r(t) = \sqrt{\alpha^2 + 2\beta t + t^2}$$

where $\alpha = |\vec{r}_O - \vec{o}|$ and $\beta = r_d \bullet (\vec{r}_O - \vec{o})$. We must compute

$$\tau = \int_0^{t_e} Ae^{-B(a^2+2\beta r+t^2)}dt$$

A solution to this integral is given by

$$\tau \approx \frac{A\sqrt{\pi}e^{-B(\alpha^2-\beta^2)}}{2\sqrt{B}}\left(erf\left(\sqrt{B}\,(t_e+\beta)\right) - erf\left(\beta\sqrt{B}\right)\right)$$

Unfortunately, the more physically plausible GADD $\sigma = e^{-r}$ cannot be reduced to such an expression, due to the complex dependence of r on t. Thus we may require a numerical integration method, ideally one that is tailored specifically to the

Team LRN

hyperbolic-quadratic exponential form of the equation and specific accuracy needed for rendering. We discuss this in a later section; for now we digress to scattering.

A MINIMAL RAYLEIGH SCATTERING APPROXIMATION

We require at least a first approximation of Rayleigh scattering to obtain proper coloration of the atmosphere. Single Rayleigh scattering adds blue light along an illuminated optical path; this is why the sky is blue. Rayleigh outscattering along the optical path reddens light coming from the background, causing, for instance, sunsets to be red. Similarly, direct light flux available for scattering is reddened; this effect is what often makes sunlight yellow. Although more elaborate and accurate models are available (Klassen 1987; Nishita et al. 1993), we have found the following extremely simple approximation to be sufficient for first-order visual realism. This model is responsible for all the atmospheric coloration effects seen in the figures from Chapters 14–19.

In our discussion so far, τ has been treated as a scalar value. In wavelength-dependent scattering, τ is a vector over wavelength λ, each λ sample having an independent value of τ. For Rayleigh scattering, τ is proportional to λ^{-4} (Klassen 1987). Each component of the τ vector requires a separate evaluation of the $e^{-\tau_\lambda}$ expression of Beer's law. Time complexity then varies linearly with the number of λ samples. The tristimulus nature of color vision dictates a minimum of three values for full-color images; hence the familiar triad of RGB samples. Larger numbers of samples, taken in the CIE XYZ color space, may yield more accurate colors (Hall 1989).

We simply note that expanding τ to an RGB vector can give a computationally minimal and visually pleasing approximation of Rayleigh scattering (see Figures 18.3 and 18.4). Correct discrete numerical integration of single scattering along the optical path requires raymarching with computation at each sample of the extinction of the direct illumination available for scattering. In the absence of such a correct but expensive scheme, extinction due to Rayleigh scattering along the optical path can be approximated by using a yellow-brown (smog-colored) atmosphere. In Figure 18.4 the RGB color of the atmosphere is `(1.0, 0.65, 0.5)` and τ = `(3.0, 7.5, 60.0)`. Observe that the atmosphere is blue against a black background, and a white background is filtered to orange, as illustrated in Figure 18.5. Figure 18.4 illustrates that the horizon still appears white against a black background, largely due to psychoperceptual reasons—it is actually no more white than the smoggy atmosphere color. We recommend this simplest model for maximally efficient rendering.

Team LRN

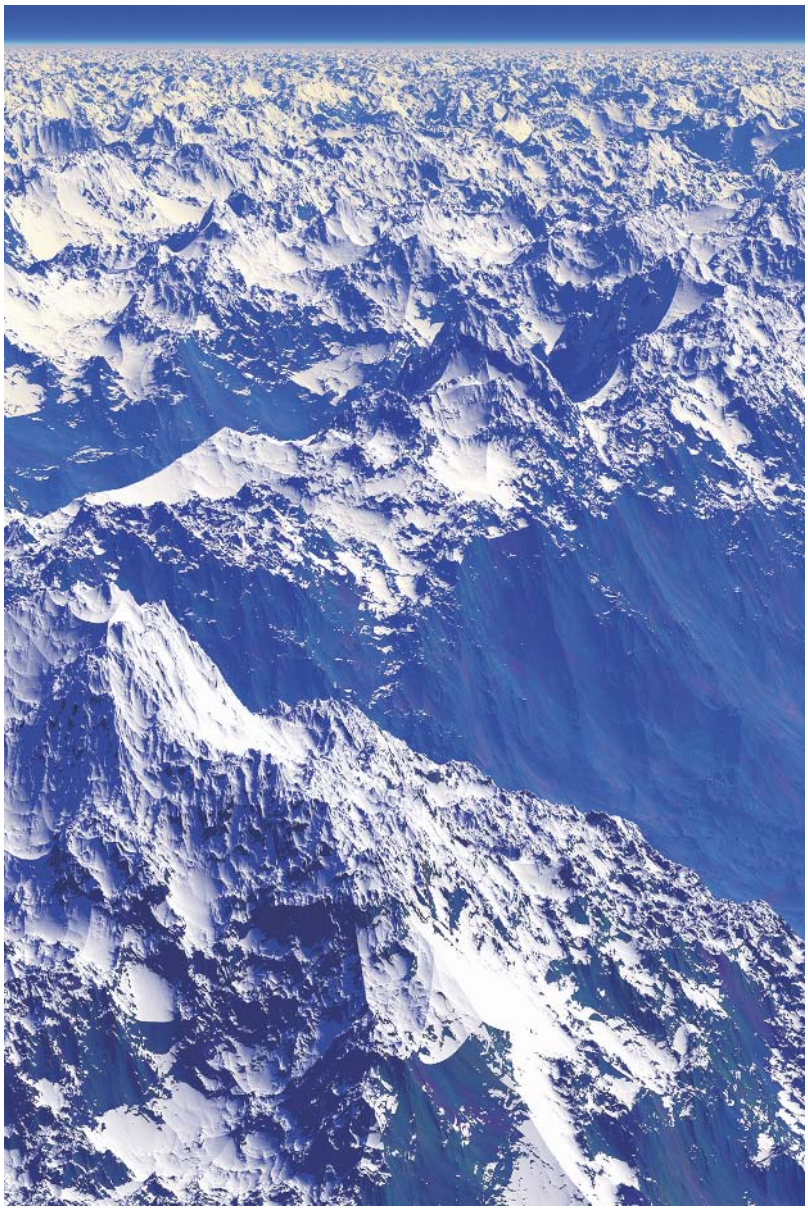


FIGURE 18.3 *Himalayas* shows color perspective over a long distance. Note that the white peaks turn red, while the dark areas turn blue with distance. Copyright © F. Kenton Musgrave.

Team LRN

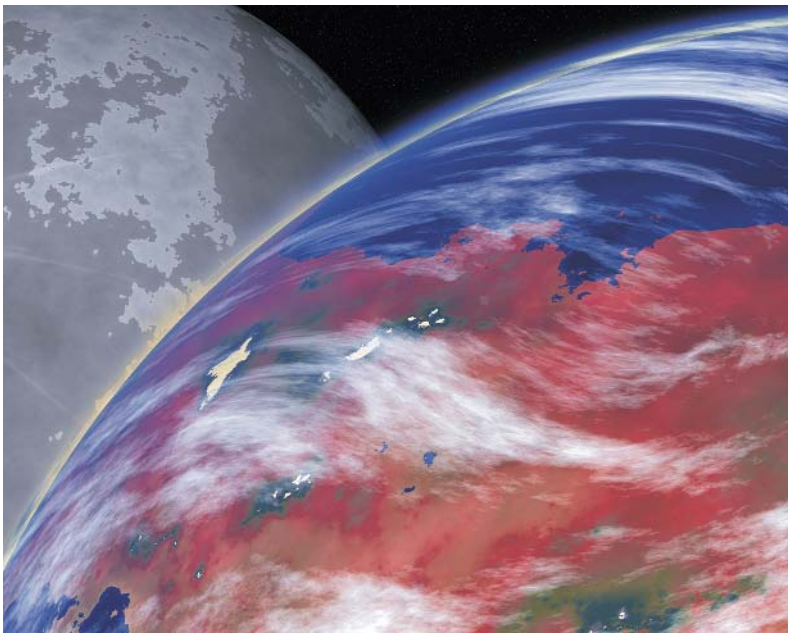


FIGURE 18.4 A detail of Figure 20.4 illustrates color perspective in a planetary atmosphere. This is the e^{-r^2} atmosphere model. Copyright © F. Kenton Musgrave.

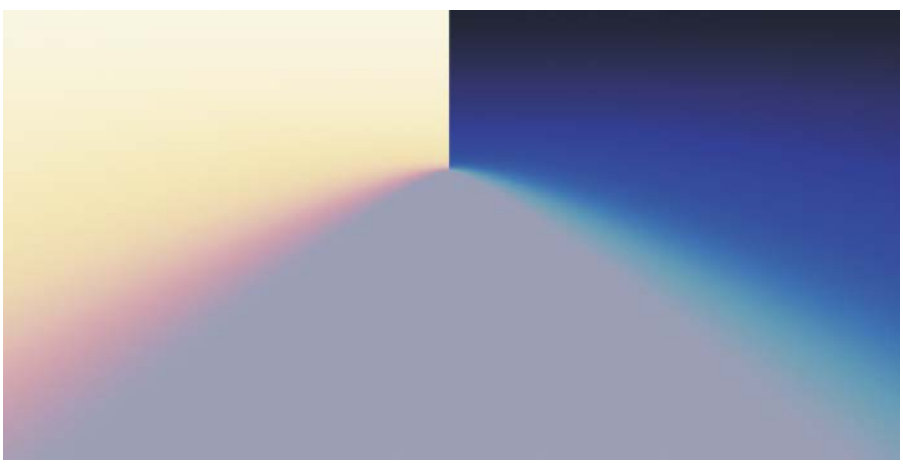


FIGURE 18.5 Color perspective in raw form. Two parallel vertical planes, one black and one white, illustrate how color changes with distance. The atmosphere is an exponential mist. Copyright © F. Kenton Musgrave.

Team LRN

To illustrate the simplicity of these two models, we now present pseudocode for the $\sigma = e^{-r^2}$ GADD with our Rayleigh scattering approximation:

```
/* compute distance from ray origin to closest approach */
adjacent_leg = ray.origin - fog.origin;
beta = DOT(ray.dir, adjacent_leg);
nearval = erf(sqrt(B) * beta);
farval = erf(sqrt(B) * (beta + t_e));
/* compute distance from fog origin to ray's closest approach */
r_c_squared = DOT(adjacent_leg,adjacent_leg) - beta*beta;
/* compute scattering approximation */
T = exp(-tau.red);
red = endpoint.red*T + (1.0-T)*atmosphere.red;
T = exp(-tau.green);
green = endpoint.green*T + (1.0-T)*atmosphere.green;
T = exp(-tau.blue);
blue = endpoint.blue*T + (1.0-T)*atmosphere.blue;
```

TRAPEZOIDAL QUADRATURE OF $\sigma = e^{-r}$ GADD AND RENDERMAN IMPLEMENTATION

For GADDs that are not integrable analytically, we must develop numerical quadrature (i.e., integration) schemes. The quadrature method may be arbitrarily sophisticated, according to the required accuracy. We now present an adaptive trapezoidal quadrature with sufficient accuracy for visual purposes and a RenderMan implementation of it.

Perhaps the most accurate, simple radial GADD is

$$\sigma(r) = Ae^{-Br}$$

where A modulates density at sea level, B is the falloff coefficient, and r is the height above sea level. For exponential GADDs, the density and its rate of change are greatest close to the planet surface. This indicates quadrature (integration) with an adaptive step size. Step size should be inversely proportional to the local magnitude of σ. Thus where the density and its rate of change in density are high, the step size is small; where the density is low, the step size is relatively large. To speed up rendering, we can employ a trivial reject if the ray never comes within a minimal distance to the GADD center, returning zero when we know the integral must be very small. Also, we can integrate separately forward and backward from the point of closest approach; this may increase accuracy through preventing overly large step sizes, by basing step size on the interval end of higher density. At each step we sample both the GADD and the illumination at the sample point.

Team LRN

Trapezoidal integration implies an assumption that the GADD varies linearly between the samples. The optical depth of a given interval is then

$$\tau = \frac{\Delta}{2}(\sigma_i + \sigma_{i-1})$$

where Δ is the step size, and σ_i and σ_{i-1} are the current and previous GADD density values. Differential extinction

$$dO = 1 - e^{-r}$$

and scattering

$$dC = I(1 - e^{-r})$$

where I is the direct illumination intensity at the sample point.

These differential values are then accumulated similarly to Drebin, Carpenter, and Hanrahan (1988). The RenderMan implementation operates in such a way that the atmosphere is shadowed where the light source is occluded by mountains, the planet, and so forth. If occluding features may be small, you must specify suitably low upper bounds on step sizes to prevent undersampling of shadow features.

Our implementation is as a volume shader in the RenderMan shading language (Upstill 1990). A simplified version of the shader follows. In Figure 18.6 a similarly structured light source shader causes the local illumination to undergo proper extinction. This and the full, more accurate volume shader are available on this book's Web site (*www.mkp.com/tm3*).

```
/* For ray index t, return the GADD (g) and illumination (li). */
#define GADD(li,g) \
      PP = origin + t * IN; \
      g = (density * exp(-falloff*(length(PP)-1))); \
      PW = transform ("shader", "current", PP); \
      li = 0; \
      illuminance (PW, point(0,0,1), PI) { li += Cl; }

volume radial_atmosphere (float density = 3.0, falloff = 200.0;
    float integstart = 0.0, integend = 10.0, rbound = 1.05;
    float minstepsize = 0.001, maxstepsize = 1.0, k = 35.0;)
{
    float t, tau, ss, dtau, last_dtau, te;
    color li, last_li, lighttau;
    point origin = transform("shader", P+I);
    point incident = vtransform("shader", -I);
    point IN = normalize(incident);
```

Team LRN

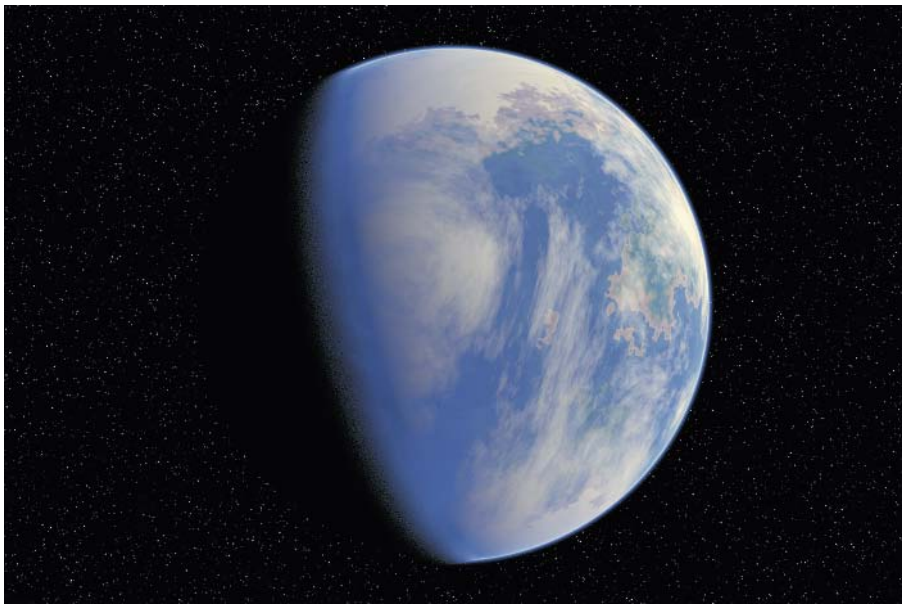


FIGURE 18.6 A raymarched e^{-r} planetary atmosphere model, implemented as a RenderMan shader. Such a model can yield more realistic coloring by including color extinction in the illumination. Copyright © F. Kenton Musgrave.

```
point PP, PW;
color Cv = 0.0, Ov = 0.0;  /* net color & opacity over optical path */
color dC, dO;              /* differential color & opacity */

/* compute optical path length */
te = min(length(incident), integend) - 0.0001;
/* integrate forward from the eye point */
t = integstart;
GADD(li, dtau)
ss = min(clamp(I/(k*dtau+0.001), minstepsize, maxstepsize), te-t);
t += ss;
while (t <= te) {
    last_dtau = dtau; last_li = li;
    GADD (11, dtau)
    /* compute dC and dO, the color and opacity of the portion
     * of the interval covered by this step */
    tau = 0.5 * ss * (dtau + last_dtau);
    lighttau = 0.5 * ss * (li*dtau + last_li*last_dtau);
    do = 1 - color (exp(-tau), exp(-tau*2.25), exp(-tau*21.0));
    dC = lighttau * dO;
```

Team LRN

```
        /* Adjust Cv and Ov to account for dC and dO */
        Cv += (1.0-Ov)*dC;
        Ov += (1.0-Ov)*dO;
        /* Select next step size and take a step */
        ss = min(clamp(I/(k*dtau+.001), instepsize, maxstepsize), te-t);
        ss = max(ss, 0.0005);
        t += ss;
    }

    /* Ci & Oi are the color and opacity of the background element.
     * Cv & Ov are the color and opacity of the atmosphere along the viewing ray.
     * Composite them together.
     */
    Ci = 15.0*Cv + (1.0-Ov)*Ci;
    Oi = Ov + (1.0-Ov)*Oi;
}
```

NUMERICAL QUADRATURE WITH BOUNDED ERROR FOR GENERAL RADIAL GADDs

The previous integration method for the e^{-r} GADD capitalizes on the smooth character of that GADD. As we may sometimes desire an error-bounded integration scheme suitable for more general radial GADDs, we now present such a scheme. As τ may be evaluated for every ray in a rendering, we require efficiency in its computation. Our knowledge of the integrand may be used to derive a suitable algorithm. Consider the general radial GADD $\sigma(r(t))$, where

$$r(t) = \sqrt{\alpha^2 + 2\beta t + t^2}$$

We can simplify by completing the square:

$$r(t) = \sqrt{(t - t_c)^2} + r_c^2$$

where r_c is the value of r at the closest point on the line containing the ray to the center $\vec{o}$ of the radial GADD, and t corresponds to the t_c index at this closest approach. (This t_c might lie beyond the ray's extent on that line.) We can then make the substitution $s = t - t_c$ and rewrite the integral:

$$\tau = \int_0^{t_e} \sigma\left(\sqrt{\alpha^2 + 2\beta t + t^2}\right) dt = \int_{t_e}^{t_e - t_c} \sigma\left(\sqrt{s^2 + r_c^2}\right) ds$$

The integrand is symmetric about $s = 0$. If the limits of integration straddle 0, this symmetry can reduce the work of the numerical integrator by up to half. We can

also specify a bounding radius for the atmosphere, that is, where the integral of an infinite optical path ultimately maps to a luminance value of zero. Let us specify this radius as $r_{\max}$. Since

$$r = \sqrt{s^2 + r_c^2}$$

the bound is

$$s = \pm\sqrt{r_{\max}^2 + r_c^2}$$

We can also trivially set $\tau = 0$ if $r_c > r_{\max}$.

We are ultimately computing a transparency value that is in turn used to compute a quantized luminance sample. For 8-bit quantization, an error in e^{-r} of less than $\varepsilon = \pm \frac{1}{512}$ is insignificant. The symmetric integrand and high error tolerance allow for a specialized adaptive integration routine. An outline of a simple algorithm of this type follows; it returns a guaranteed bounded estimate for any radial GADD that decreases monotonically with radius.

1. Compute distance of closest approach r_c and ray index t_c at r_c.
2. If $r_c > r_{\max}$, return 0. Let radial cutoff bound

   $$s_m = \sqrt{r_{\max}^2 + r_c^2}$$

   If $s_m > t_c$, return 0. Compute integration bounds a and b: set $a = -\min(t_c, s_m)$. Set $b = -\min(t_e - t_c, s_m)$; if $b < -s_m$, return 0.
3. If $a < 0$ and $b > 0$, define two "regions" $R^{[1,2]}$. Region $R^{[1]}$ has left bound $R_l^{[1]} = 0,$ right bound $R_r^{[1]} = \min(-a,b)$, and a weight R_w of 2.0; region $R^{[2]}$ has left and right bounds $R_l^{[2]} = \min(-a,b)$, $R_r^{[2]} = \max(-a,b)$, and a weight $R_w = 1.0$. Otherwise, make a single region with $R_l = a$ and $R_r = b$, with $R_w = 1.0$.
4. Compute the values of σ at left and right region bounds; label them $R_{\sigma l}$ and $R_{\sigma r}$. The maximum error $R_{\delta\tau}$ for the region, assuming Euler integration (i.e., the worst case), is $|R_{\sigma r} - R_{\sigma l}|(R_r - R_l)R_w$. The trapezoidal estimate for the region's integral R_τ is $\frac{1}{2}(R_{\sigma l} + R_{\sigma r})(R_r - R_l)$.
5. The total error

   $$\delta\tau = \sum\nolimits_R R_{\delta\tau}$$

Team LRN

The total integral estimate

$$\tau = \sum_R R_\tau$$

If

$$\left(e^{\delta\tau-\tau} - e^{-\tau}\right) \leq \varepsilon$$

return τ.

6. Find the region with the highest error estimate. Bisect that region; one evaluation of σ is required. Compute integral and error estimates for both new regions. Go to step 5.

Potential modifications to this algorithm include splitting several regions at a time to allow less frequent checks of the error criterion and using Simpson's rule for integration.

CONCLUSION

We have presented a series of GADD models of increasing fidelity to the geometry of a planetary atmosphere. We have suggested a Rayleigh scattering model at a level of fidelity comparable to the ambient/diffuse/specular surface illumination model (i.e., highly nonphysical but simple, intuitive, and useful). We have illustrated the successful use of these models in image synthesis. Occam's Razor may recommend these models due to their simplicity; indeed, some are simple enough to be candidates for hardware implementation in real-time graphics systems. RenderMan implementations of these models are available on this book's Web site (*www.mkp.com/tm3*).

Team LRN

19

GENETIC TEXTURES

F. KENTON MUSGRAVE

INTRODUCTION: THE PROBLEM OF PARAMETER PROLIFERATION

As we saw in Chapter 15, one problem confronting us in the construction of procedural textures or shaders is that of *parameter proliferation*. The "terran" texture presented in Chapter 15 has 22 user-definable parameters, plus some 51 hard-coded constants hidden in the shader code. The results of changes to these parameters are often far from obvious and sometimes downright bizarre. For use in a production environment by artists who didn't write the shader, this kind of situation is simply absurd. Most users of shaders—digital artists—don't know about, or want to be confronted with, the rich logical complexity and odd machinations of the shader's internal operation. They generally have another job to get done: making pictures, within rigid constraints on time and quality. It is unfair and counterproductive to require them to learn about or necessarily understand how shaders work. And yet the terran texture illustrates how, in fact, shaders are built and operate.

On the other hand, neither is it all that easy for the programmer or shader writer to define and implement all those parameters. It's tedious, arduous, and generally time consuming to write and refine such complex shader code. The ultimate product that the programmer should ideally deliver is a simple, intuitive user interface that's easy for an artist, with no programming or math background, to understand and use. Unfortunately, if effective shader writing is a black art, then devising such interfaces is a black hole.

I hate to sound defeatist here, but ever since Gavin Miller first pointed out this problem to me in 1988, when he was working at what is now Alias/Wavefront, I have been bemused by this problem. I have not met anyone who has an effective, general strategy for reducing a huge number of parameters to a few intuitively obvious sliders that maintain the power of the underlying functionality. I obtained direct experience in managing this problem when producing the Disney Cruise Lines commercial, where Mickey and Goofy are made out of clouds, at Digital Domain in the spring of 1997. In that case, we—the art director and myself—simply revealed

Team LRN

parameters to the artists one at a time, as we perceived that they needed them or that the look would benefit from their knowing about them. The vast majority of parameter values were preset by myself, the programmer. Unfortunately, this requires that the programmer also be something of an artist, something that is not always possible to achieve in a production context.

A USEFUL MODEL: AESTHETIC *n*-SPACES

Here is one way of thinking about the textures and their controls that I find useful: they represent an *aesthetic n-space*. The "aesthetic" part simply means that changing values of the parameters affects the aesthetics of the result. The "*n*-space" part is more subtle. The "*n*" in "*n*-space" is simply some whole, positive number, from zero to infinity. Each separate one of those *n* numbers represents a *degree of freedom*. Think of a degree of freedom as a new direction in which we can move. For $n = 1$ we have a line and exactly one axis along which we may move in two directions, call them left and right, for convenience. For $n = 2$ we have a plane, wherein we may move left and right and, say, forward and backward. For $n = 3$, we have the familiar three-space in which we live, wherein we may move left and right, forward and backward, and up and down. When *n* equals 4 or higher, we move into the higher dimensions for which human intuition fails us but into which mathematicians never hesitate to go.

In this model, the terran texture presented in Chapter 15 represents a 73-dimensional aesthetic space! No wonder you wouldn't want to hand that shader over, as written, to a digital artist. Believe me, it took more than a few hours to define and determine values for those 73 parameters, too. How can we deal with these two problems, the overwhelmed user and the overworked programmer? *Genetic programming* can provide a fascinating solution to both. But before I describe exactly what genetic programming is, let me first describe some motivating concepts behind our process.

The process of defining the *n* parameters in a procedural texture corresponds to the *creation* or *specification* of the *n*-space. The process of determining good values for the parameters may be thought of as *searching the n-space for local maxima of an aesthetic gradient function*.[1] This is an abstraction of which I am particularly fond: as we change the values of the parameters, we move about in the *n*-space, in a manner exactly analogous to the low-dimensional spaces described earlier. As we move

1. This model is based on what are called hill-climbing optimization methods, such as simulated annealing (Press et al. 1986).

Team LRN

about, the aesthetics change. How they change is determined by an entirely subjective aesthetic judgment on the part of the user. But clearly some sets of values will provide images that are "better" and other images that are "worse." The aesthetic gradient function is then the user's subjective evaluation of how the "goodness" of the result changes with changes in the parameter values, the gradient being between "better" and "worse." A local maximum represents a set of parameter values where, if any one is changed a little, the image gets worse. Thus we're in a position analogous to being on a local hilltop or local aesthetic maximum. Small movements in all directions in *n*-space correspond to moving downhill in terms of our aesthetics. Yet this hilltop is only local—there is no guarantee that, if we move far enough away from our current point, we'll cross the equivalent of some "aesthetic valley" and be able to climb up a higher hill to a better local aesthetic maximum. The nice thing about this model of the creation and search of *n*-space is that it is independent of the value of *n* and therefore of the complexity of the texture or shader.

CONTROL VERSUS AUTOMATICITY

An inevitable outcome of the growth of complexity (e.g., the number of parameters) is that there arises an eternal tension that is general to models for image synthesis: control versus ease of use. If you make things clear, simple, and easy for the user, you necessarily have to compromise control, because control lies in the complexity of the procedures. If you give the user full control, the interface becomes overwhelming in its baroque complexity. Anyone who's used 3D modeling or rendering software, from low-end consumer to the top-of-the-line professional packages, has confronted this problem.

As we've tried to make clear through much of this book, the whole paradigm of proceduralism is intimately caught up with this idea of *amplification,* described in Chapter 14, whereby lots of visual detail issues from a relatively small number of controls (parameters). Unfortunately, the flip side of this wonderful power is that automaticity implies lack of control. Just as in any human project large enough to require delegation of subtasks to colleagues, you abdicate full control over the results. Thus we may construct the beautiful planet Gaea, imbued with the capacity to be imaged at any range and/or field of view and resolution (see Figures 16.3 and 16.6), from a very small amount of computer code, but we cannot, without compromising elegance, control any of the specific features found there. We have only qualitative controls with global effects.

Nevertheless, we can obtain some striking and useful results. But what if we take this amplification/automaticity to its logical extreme and let the computer do

Team LRN

everything? Then the user would simply sit back and pick and choose from various offerings, like Scarlet O'Hara selecting a beau for a dance. In this paradigm, the computer simultaneously specifies and searches the aesthetic *n*-space. The method is spectacularly productive, wonderfully automatic—once a lot of programming has been done—but difficult to direct to a desired end, for example, a wood-grain texture. From a practical point of view, this last point may be a fatal flaw. Yet I'll describe the genetic programming paradigm here because it illustrates the functional nature of procedural textures and clarifies through extreme abstraction how such textures are built, and simply because genetic programs are the most fun software systems I've ever played with. Because the details of implementation are tedious, but the concepts driving them are quite clear, I'll stick to a high-level description here. The code for the genetic program called Dr. Mutatis that I wrote for MetaCreations went the way of that ill-fated company and is no longer available, unfortunately.

One thing you might keep in mind when reading this chapter is my basic motivation for taking on the nontrivial task of programming genetic textures: breeding planets. You see, since 1987 I've been working toward building a synthetic universe. One big problem: a universe has a *lot* of planets—far too many to build by hand with code as complex as the terran texture. The solution? Have the computer build them for us, automatically—or as close to "automatically" as we can get. The synthetic universe we're out to build is meant to be full of surprise and serendipity, so exact control of all the details is not important. We're not out to build a preconceived stage set, but rather to explore some of the beauty inherent in mathematics and logic, the mathematics and logic embodied in the texture code. This gets rather philosophical and is covered in the final chapter of this book. For now let's jump into the genetic approach to building procedural textures.

A MODEL FROM BIOLOGY: GENETICS AND EVOLUTION

Genetic programming starts with a model borrowed from biology and proceeds to use it by analogy. The idea was introduced to the computer graphics community by Karl Sims in his 1991 SIGGRAPH paper (Sims 1991). Sims in turn got the idea from Richard Dawkins's book *The Blind Watchmaker* (Dawkins 1987) and the simple computer graphics program called BioMorph that Dawkins uses to illustrate the power of the theory of evolution in explaining the origins and complexity of life on Earth. Unlike Dawkins, I have no metaphysical ax to grind vis-à-vis the origin of life or competing religious and scientific models for the origin of life. I simply have found genetic programming to be the coolest thing you can do with procedural

Team LRN

textures and computer art and an important stepping stone toward the future of proceduralism that I envision, this synthetic universe.

We start with a few definitions that should be familiar from your high school biology classes. Recall that the *genotype* is the genetic description for a given organism. The genotype is encoded in a fantastically long molecule of *deoxyribonucleic acid* (DNA). The genotype is a specific instance from a *genome,* as in the Human Genome Project, the major scientific initiative that has mapped the general layout of all human genes. The genome is general to a species and has variations among individuals, while the genotype is specific to a single organism. A *gene* is a specific part of the genome that encodes a certain function, generally instructions for building a certain biologically active protein.

The *phenotype* is the physical manifestation of the instructions encoded in the genotype: it is a specific organism, such as you or I. Your genotype is similar to mine, but they are not identical, so while we are both human beings, we are not identical twins or clones. Different instances of a given genotype will, given similar environments during development, reliably give rise to a certain, well-defined phenotype, as with identical twins (who are, in fact, clones).

Charles Darwin's famous and controversial theory of evolution posits that life has literally risen from the primordial ooze through *evolution*—the refinement of genomes through what he called *natural selection:* the preferred survival and propagation of individuals whose genotype has given rise to a "more fit" phenotype.[2]

Natural selection acts on phenotypes. No progress would occur if the phenotypes didn't change over generations. In nature they do, by two mechanisms: *mutation* and *sexual reproduction.* Mutation occurs through errors introduced in the DNA replication process and by direct alterations to DNA molecules by external mechanisms such as ionizing radiation (for example, ultraviolet light and radioactive decay byproducts). Sexual reproduction is presumed, in biology, to be a clever adaptation by higher organisms. In sexual reproduction, genes from two parents are mixed and matched to "reshuffle the deck," providing random combinations of proven genes. This is a safer strategy for productive change than the purely random variations provided by mutation, most of which presumably will not produce viable phenotypes.

2. Note that Darwin's theory of evolution preceded the discovery of DNA by about a century. The theory of evolution is predicated only on the idea of *inheritance,* whereby offspring acquire the genetic information of their ancestors. It is in no way dependent on the mechanism for encoding or passing on that information. This independence helps bolster the analogy we're making here.

Team LRN

Evolution is the accumulation of "improvements" in the phenotypes through these changes, as culled by natural selection, in inheritable genomes. This process can be convincingly argued to account for all the glorious complexity and variation in life on Earth, as Dawkins's series of books on the topic attempts to do.

I have come to think of DNA as being like the operating system (OS) of an organism, while the cells of which the organism is composed are like the computer on which the OS runs. (Fortunately, the human OS is generally more reliable than those we've devised for our computers.) The coding of both DNA and an OS is very abstract: the program code that comprises a computer OS bears little resemblance to the user interface it presents to us, just as the DNA molecule little resembles a hamster, a redwood tree, or you. Also similarly, both are highly nonportable: the encoding is practically worthless without the platform on which it is designed to execute. Hence you can't just install and run the Mac OS on a PC, nor can you put human DNA into a starfish cell nucleus and expect to grow a healthy human baby. This is a little unfortunate for our application here, as we'd like to have a universal computer genome that could be run on any computer, so that we could develop a kind of universal artificial life, or "A-life" as it's called.

The Analogy: Genetic Programming

What we're interested in here is procedural textures and how to create beautiful ones efficiently. Enter our analogy: We will regard the code that specifies a texture to be its genotype and an image of the resulting texture to be the corresponding phenotype. Evolution is directed by what I call *unnatural selection*:[3] God-like intervention by the user, deciding which phenotypes, and thus the underlying genotypes, survive and propagate. Change in the genome is accomplished by methods analogous to mutation and sexual reproduction: we design the program to introduce random variations in genotypes and to be able to share "genes" in an analog of sexual reproduction.

What then, is a gene in a genetic texture program? As with DNA, it is a unit of genetic "code" specifying some functionality within the resulting texture, for example, an fBm procedure. DNA is composed of the four nucleic acids cytosine, guanine, adenosine, and thymine, commonly referred to by their initials C, G, A, and T. The

3. I like this tongue-in-cheek term "unnatural selection" because it points out the artificial separation of humankind from nature. In my view, humans and their actions are natural phenomena. If you disagree, I suggest you try—in a thought experiment—separating humans entirely from nature and see how we do!

Team LRN

DNA molecule is a long sequence of these *bases,* as they are called, paired across from one another in the famous double helix. A certain functional sequence of bases can comprise a gene.

Our encoding scheme is a little different. Our bases are all complete functions, analogous in DNA more to genes than to bases. A combination of our bases can, however, function as a gene. The difference between genes and bases is that bases are *atomic:* bases cannot be subdivided into smaller parts.

The encoding scheme for our genetic information is, rather than a linear sequence as in DNA, an *expression tree,* which is analogous to a genealogical family tree. (See Figure 19.1.) A tree is a special kind of graph. The graph is composed of *nodes*. There are two relevant kinds of relationships between nodes: *parent* and *child,* the meaning of which is obvious. There are three types of nodes: the *root* node at the top of the tree (although it might seem that it should be called the bottom), which has no parent and usually has children; *interior* nodes, which have both parents and children; and *leaf* nodes, which have parents but no children. The root node generally has to return three values to create an RGB value to display as the phenotype.

The expression tree operates via *functional composition,* described in Chapters 2 and 15 as *perturbation* or *domain distortion*. The idea of functional composition is simply that a function takes as its input parameters the output of another function or functions. We saw the effects of simple functional composition in those earlier chapters; now we take the idea to an extreme. In the expression tree, only leaf nodes

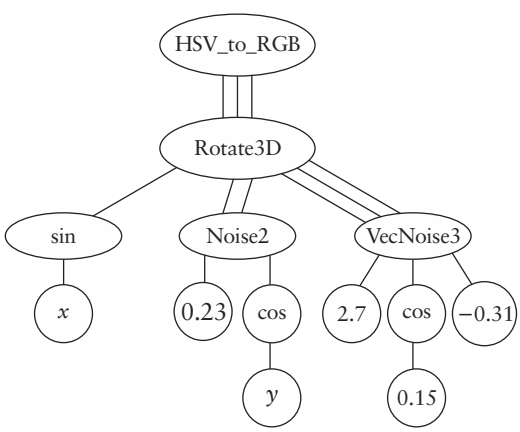


FIGURE 19.1 An expression tree. The circles are nodes; the lines between them are *links* representing relationships.

provide values that are not determined by functions. (In fact, the leaf nodes are usually simple linear functions of *x*, *y*, or *z*; that is, they are simply the *x*, *y*, or *z* value of the point where the texture is being evaluated. The rest are simply random numbers.)

Expression trees "evolve" via random mutation and sexual reproduction, as illustrated in Figures 19.2 and 19.3. The user selects which phenotypes mutate and breed.

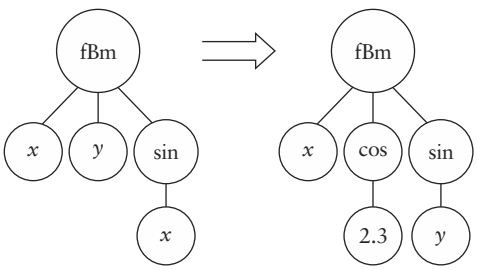


FIGURE 19.2 Mutation in an expression tree.

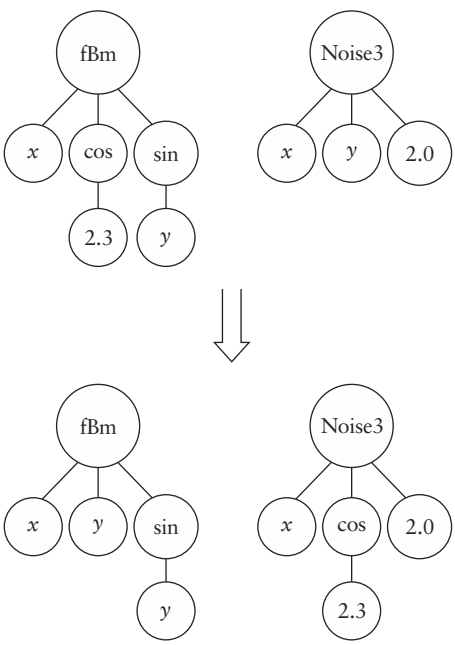


FIGURE 19.3 Sexual reproduction between two expression trees.

Implementation

The expression tree is perhaps most easily implemented in a high-level, functional language such as LISP, as in Sims's original genetic texture program. Unfortunately, LISP is relatively slow, unless you happen to have something like a Connection Machine 2 to run it on. Indeed, my first experience with genetic textures involved running Karl's LISP code on a CM-2 using 16,000 processors. Ordinary users like you and me will require a more efficient implementation for our more pedestrian computers. In fact, I developed my genetic texture code primarily on a laptop, showing how far processors have come since 1991. Steven Rooke tells me that switching from LISP to C++ sped his genetic program by a factor of 100. (And yes, I know LISP fans who'd contest that!)

It turns out that genetic programming is a perfect application for C++. The data structures required for the various types of nodes, their relationships, and the operations defined upon them are succinctly described in C++ classes. The devil is in the details, mostly in memory management. For efficiency, I have each evaluation of a function in the tree process an entire scanline's worth of data. (This saves a whole lot of tree traversals that would be necessary if you evaluate the tree pixel by pixel.) For large trees, this can get into a lot of memory. Again for efficiency, I do all my own memory management in the program because the C and C++ memory allocation routines `malloc` and `new` are relatively slow. So I end up with piles of pointers and memory pools—and lots of room for bugs. But that's just standard programming and thus outside the scope of this book.

There are two interesting issues in programming genetic textures that I'd like to point out: the meaning of the root node, in terms of color, and the effect that a given library of genetic bases has on the kinds of images produced.

INTERPRETATION OF THE ROOT NODE

Ultimately, we want to make color images. Thus each pixel will require a separate value for red, green, and blue. A solution that immediately pops into mind is that we simply have the root node consist of three separate subtrees, one each for red, green, and blue. In practice, however, this is usually unsatisfactory: you tend to end up with unrelated, overlaid images in red, green, and blue. You can interpret the values as lying in another color space, such as HLS (hue/luminosity/saturation), but similar problems remain. The usual solution is to have the root function return a single value that serves as an index into a color lookup table. This brings up the separate and unrelated problem of generation of, and making changes to, that lookup table. It is easy enough to automatically generate random color maps (Musgrave 1991),

Team LRN

but the obvious solutions are a little inelegant compared to the rest of our fully procedural paradigm.

The solution I'm currently using is based on the ideas behind the random color textures presented in Chapter 15. The root node comprises a three-vector valued function, which has been passed through a random rotation matrix to correlate, in the final RGB color space, the influence of the three components of the vector. Mathematically, we'd say that the three vector components are then *linear combinations* of the *basis vectors* that correspond to red, green, and blue. That is, rather than having each component of the three-vector mapping to only red, green, or blue, each component contributes to all of red, green, and blue, albeit indirectly through a final HLS to RGB transform (see the expression tree in Figure 19.1). From a mathematical perspective this insight is obvious; my apologies if it's not exactly clear when translated into English prose. Such is the divergence of the two modes of thinking and communication. But it's cool to see again, as in the GIT schemes described in Chapter 15, that a mathematical perspective can provide useful aesthetic insights.

This approach is not without its own problems. First, constructing a random rotation matrix requires at least six input values: two—altitude and azimuth—to specify the random rotation axis, one to specify the angle of rotation, and three to specify the vector being transformed by the resulting matrix. This implies a bushy tree at the root, which in turn implies increased evaluation time. It also implies a rather large amount of storage in the vector class—a maximum of six double-precision floating-point numbers,[4] which adds up when you need to store a lot of vectors. This approach also tends to consistently produce rainbows, due to the final HLS to RGB transform. These rainbows become boring to annoying but can be excised through the process of unnatural selection. The advantage of the approach, as I see it, is that it nicely preserves the pure functional paradigm.

THE LIBRARY OF GENETIC BASES

An important aspect of any genetic image generation program is its library of bases, the functions out of which the expression trees are formed. This library literally provides the expressive vocabulary of the system. Thus different genetic texture programs have different "looks." Karl Sims's system has a library of primitive mathematical functions such as sine, log, arc tangent, and so on, as well as iterated

4. My experience indicates that single-precision floating point does not provide sufficient accuracy for the kind of multiple functional composition involved in genetic textures.

Team LRN

function systems (Sims 1991) that give rise to characteristic fractal patterns. Steven Rooke's system (*www.azstarnet.com/~srooke/*) is heavy on deterministic fractal functions that are generally iterations on the complex plane, including a genetic generalization of the kinds of functions people experiment with in the well-known Fractint freeware program. My own system (*www.kenmusgrave.com/mutatis.html*) is based primarily on the kind of random fractal functions described in Chapters 14 and 16. Thus each system tends to create images with a certain, fairly consistent character. Certainly, each produces images that the others are not capable of generating, due to the expressive limitations of their respective libraries of bases.

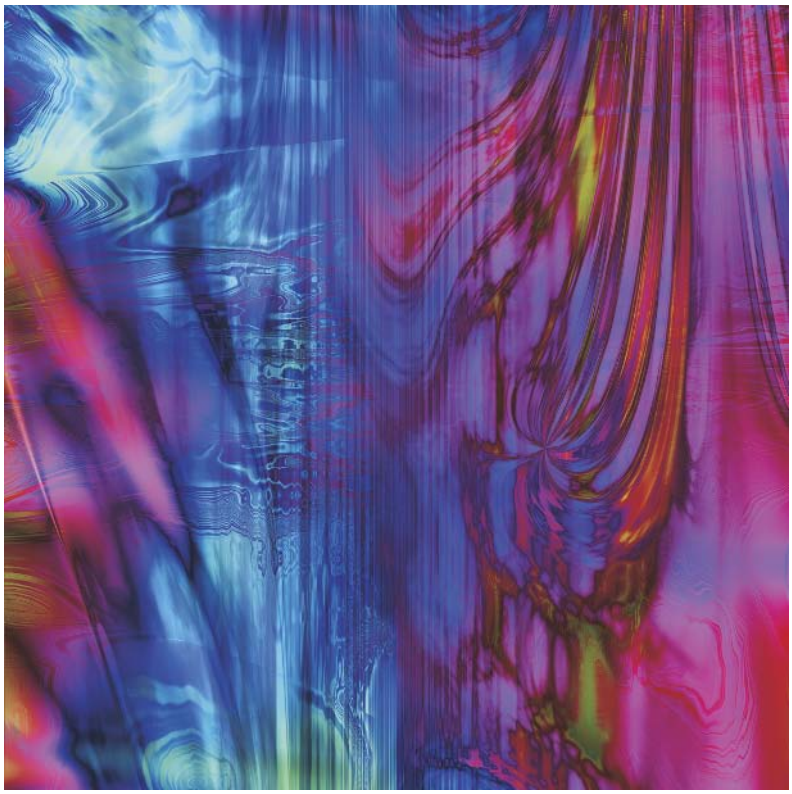
My own peculiar base functions tend to be very natural-looking, as they were originally honed for the modeling of natural phenomena such as mountains, clouds, and water. They thus tend to generate images that look like they were executed in a natural medium, such as oil paint. In fact, one of my main motivations in going into this area was to automate the generation of painterly textures such as that seen in Figure 15.17. Examples of my system's output are seen in Figures 19.4–19.9.

One nice thing is that the longer you work with a genetic program, the more "evolved" the results become. That is because as you accumulate a library of "fit" individuals, they can trade genetic information via sexual reproduction. Indeed, one feature of my program is being able to have individuals breed with the entire population of saved genomes, in a sort of orgiastic exchange of genetic information. An effect of this continued evolution is that subtrees become genes in their own right, being swapped in whole in the breeding process. Thus the "library" of genes can grow both in size and complexity as evolution proceeds. (We are now referring to a single function—what computer scientists would call a "primitive" or "atomic" function—as a base and a tree of any size as a gene.)

OTHER EXAMPLES OF GENETIC PROGRAMMING AND GENETIC ART

Karl Sims has taken this paradigm of genetic programming and genetic art farther than the rest of us. For example, he evolves the behaviors of three-dimensional textures (Sims 1991) and virtual creatures (Sims 1994) (see *www.biota.org/ksims.html*). William Latham (Todd and Latham 1993) has done some remarkable work in a system designed to generate 3D sculptures that can bear uncanny resemblances to creatures from the Cambrian epoch of life on Earth (*www. artworks.co.uk/*). Roman Verostko (*www.verostko.com*) employs the related concept of *epigenesis,* or the unfolding of form in the phenotype in the process of growth, in his plotter-generated artworks. Eric Wenger's ArtMatic product (*www.artmatic.com*) lets you play with a version of genetic textures.

Team LRN



FIGURES 19.4–19.9 Images generated by the genetic program Dr Mutatis. Note the rich visual complexity that arises through the automated evolution of a procedural texture. Copyright © F. Kenton Musgrave.

This is by no means an exhaustive listing of artists and scientists working in this exciting area. A search on the Web will turn up thousands of relevant sites. A good general guide is Linda Moss's site (*www.marlboro.edu/~lmoss/planhome*), which even offers source code. The bible of genetic programming is Koza (1992).

A FINAL DISTINCTION: GENETIC PROGRAMMING VERSUS GENETIC ALGORITHMS

In the literature, you'll read of genetic programming and genetic algorithms. The former is what I've described here, wherein the very code of the program that generates

FIGURE 19.5

the phenotype is itself transforming over time. The latter is a little different: it assumes a fixed value on *n* for the *n*-space it explores and is thus perhaps more closely related to optimization strategies than to free-form artificial evolution. That is, it only searches the *n*-space for local aesthetic maxima, while genetic programs simultaneously define and search their *n*-space, with the value of *n* constantly changing. Thus they tend to be simultaneously more chaotic, hard to control, and productive—a familiar set of characteristics among creative people! At any rate, it may be helpful to be aware of the difference, so I'm pointing it out here.

Team LRN

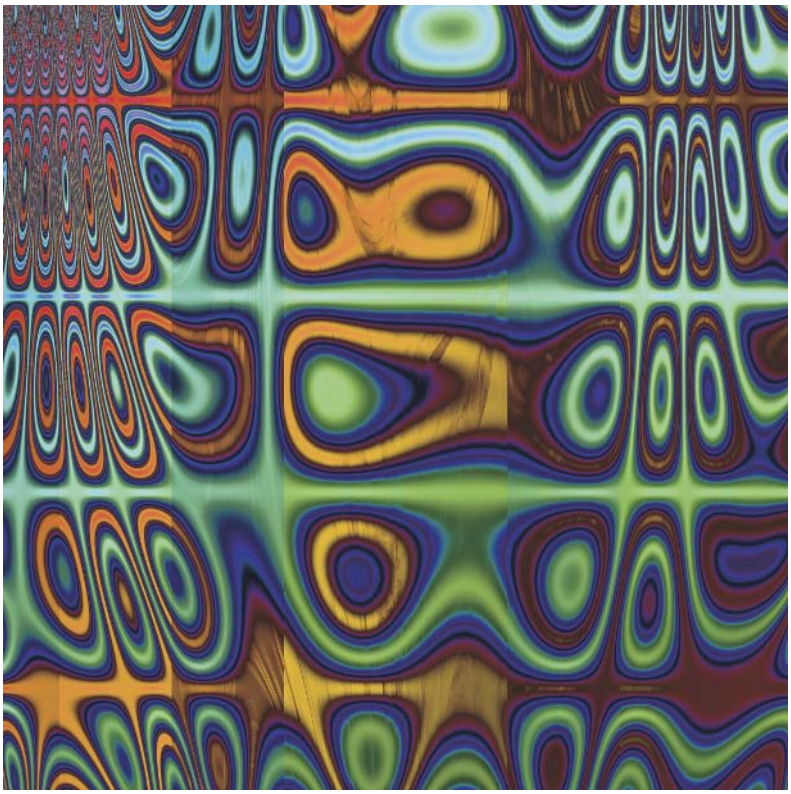


FIGURE 19.6

CONCLUSION

Genetic programming is one cool paradigm. It is amazingly automatic, has the world's best user interface—simply point and click on what you like—and it takes proceduralism to its logical end. While it may not be terribly *useful* because it's so hard to control and direct, it certainly is fun to play with, and it does create some striking images at times. It also gives the computer the greatest role in the creation of digital art of any paradigm I know. This is exciting in itself, as the computer can be a very capable, if simple-minded and cranky, artistic assistant.

Someday a program or programs will bring "genetic textures to the people." One fun thing about this prospect is that people could start trading genomes on the

Team LRN

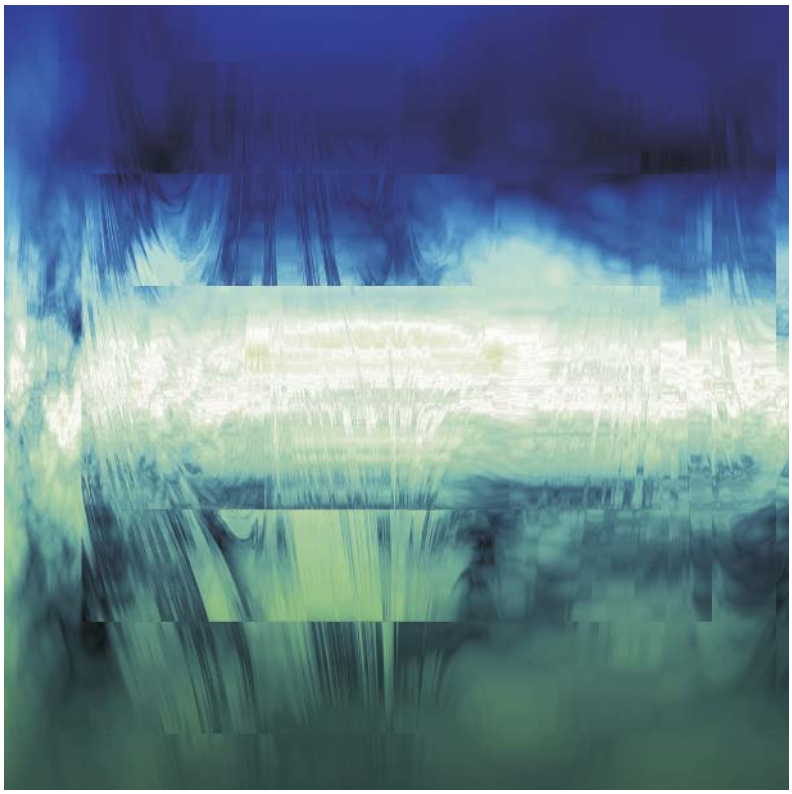


FIGURE 19.7

Internet, thus accelerating evolution by cross-breeding between populations that spend most of their time in isolation on a given user's machine. This is analogous in nature to the evolutionary divergence of populations isolated on islands, which are occasionally intermingled by migration, chance travel, or formation of a land bridge.

Such universal exchange of genetic information will require standard encoding and interpretation machinery, just as DNA from some extraterrestrial organism would have zero chance of intermingling with that of life on Earth. Again, the genotype is like the operating system, and the cell is like the computer it runs on. Genotypes and operating systems are not highly portable. I foresee that, should we succeed in bringing genetic art to the people, we will all suffer for some time to come for a lack of foresight in the design of a robust and flexible system. I hope that one

Team LRN

FIGURE 19.8

day we'll be able to "breed" entire galaxies—indeed, a whole universe—of procedural planets, replete with automatic level of detail, in a future version of MojoWorld, which we'll get into in the next chapter. But that will be a few years from now, at least. We'll need a *lot* more processor power before we can deploy the powerful but expensive technique of genetic programming in that context.

Team LRN

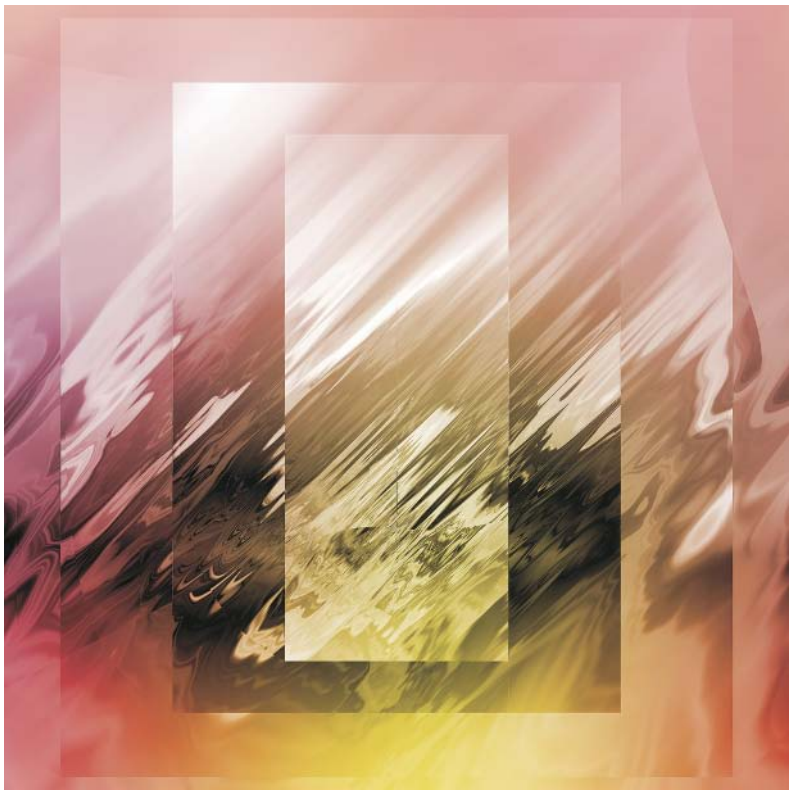


FIGURE 19.9

Team LRN

20

MOJOWORLD: BUILDING PROCEDURAL PLANETS

F. KENTON MUSGRAVE

INTRODUCTION

This chapter is derived from the first chapter of the MojoWorld manual. MojoWorld is a software product that builds and images fractal planets. These planets are entirely procedural, unless augmented with imported data. They are fractal. MojoWorld employs most of the tricks and techniques described in Chapters 14–19 of this book to create images with unprecedented detail (see Figures 20.11, 20.16, 20.17, 20.42, and 20.43). These images are sometimes realistic, sometimes surrealistic, and sometimes outright abstracts. As I've mentioned, my life's work is about revealing a synthetic universe. MojoWorld is our first cut at that.

Are we building this universe, or are we discovering it? That's a fascinating question. Sure, we won't stumble on it by accident; rather, we'll conjure it forth through a lot of intelligent and hard work. But, this "synthetic" universe already exists. It always has and always will. Just as 2 is the eternal result of the computation 1 + 1, presumably for all time and space in this universe, these places, as three-dimensional models, and the images we make of them, are the result of a more complicated but similarly deterministic calculation. They may already have been imaged and explored by other intelligent life-forms elsewhere and elsewhen in this universe. Their complexity and dependence on an enormous series of more or less arbitrary decisions by the programmers who wrote MojoWorld makes it extremely unlikely that any place or image we make/discover in MojoWorld has been seen, made, or discovered anywhere else by anyone else. But they have this eerie quality of "being there," just waiting to be discovered and explored (see Figure 20.1). This is surely the apotheosis of proceduralism.

Why include this chapter in this book? First, it explains fractals yet again. In all my years of experience as a teacher of fractals, I've found it reliably takes about three

Team LRN

FIGURE 20.1 *Metallichron: Brassy Rise* is an image from one of the planets that MojoWorld users shared, explored, and imaged in the weekly challenge at *www.3dcommune.com*. This kind of self-organizing community event is part of the MojoWorld experience. © 2002 Armands Auseklis, MojoWorld by Robert Buttery.

passes over explanations of fractals before you start to "get it." It took me at least that many times, believe me! So this chapter provides a third pass at it. Second, this chapter explains MojoWorld as a learning lab in which to experiment with fractals and procedural textures. MojoWorld is one of the most potent tools to date for that purpose, and a demo version is included on this book's Web site (*www.mkp.com/tm3*) for your pleasure. Using MojoWorld in concert with your readings in this book will turn your experience from a purely intellectual reading exercise to a hands-on practical tutorial.

An inside tip: There's a part of MojoWorld we don't talk about much—the Pro UI. It's a powerful dataflow function graph editor for writing procedural shaders—a kind of visual programming language. We don't talk about it much because it's too advanced for our target market; they'd simply be flummoxed by it. But it's perfect for anyone reading this book, as it allows you to experiment with many of the methods and tricks we've described here. And, if you find you can't do what you want

Team LRN

with the Pro UI as is, MojoWorld has an open architecture allowing you to extend its functionality by writing your own plug-ins. MojoWorld was designed to be a high-tech sandbox for all of us to play in. We encourage you to play and experiment with MojoWorld, and we look forward to seeing what you create/discover.

This chapter ties together most of what I've written in previous chapters and describes it all in the context of a software package that's included on this book's Web site for your use. It also points the way directly to my personal vision of cyberspace, the future of the human-computer interface, as described in the final chapter. So, in a very real sense, this chapter puts it all together and points to the future. I hope you enjoy it. Let's get started with another description of fractals.

FRACTALS AND VISUAL COMPLEXITY

Nature is visually complex. Capturing and reproducing that complexity in synthetic imagery is one of the principal research problems in computer graphics. In recent years we have made impressive progress, but nevertheless, most computer graphics are still considerably less complex and varied than the average scene you see in nature. Personally, I don't expect computer graphics to be able to match the visual richness of nature in my own lifetime—there's just too much complexity and variety to be seen in our universe. But that certainly doesn't mean we shouldn't try, only that we can keep at it for a long time to come.

So how do we make a first stab at creating visual complexity in synthetic imagery? In a word, with fractals. Fractal geometry is a potent language of complex visual form. It is wonderful in that it reduces much of the staggering complexity we see in nature to some very simple mathematics. I'm going to try to convey, as simply as I can, the intuition behind fractals in this chapter. I know it's a little confusing the first time around. It took me several rereadings of the standard texts when I was a graduate student to get it straight in my head. But after I "got it," it became clear that the important parts are very simple. I'm going to try to convey that simple view of fractals in this chapter. First a little motivation, though.

Building Mountains

One of the most common fractals we see in nature is the earth we walk on. Mountains are fractal, and we can make very convincing synthetic mountains with some pretty simple fractal computer programs. Benoit Mandelbrot, the inventor/discoverer of fractals and fractal geometry, calls such imitations "fractal forgeries" of nature. My own career in fractals began with making some new kinds of fractal

Team LRN

FIGURE 20.2 *Dale Stranded* illustrates the extreme depth, with pixel-level detail throughout, that can be imaged with the procedural methods MojoWorld employs. © 2002 Armands Auseklis.

terrains through novel computer programs, to create fractal forgeries the likes of which had not been seen before. Figure 20.2 shows the current state of the art in such "forgeries."

When I began to crank out some very realistic images, people immediately started asking me, "Why don't you animate them?" Well, there aren't many moving parts in a landscape. Landscapes tend to just sit there rather peacefully. What *is* free to roam about is the camera, or your point of view. This in turn begs the questions: Where do you come from? Where do you go? If the point of view is free to roam, the landscapes need to be in a proper global context. What's the global context for a landscape? A globe, of course!

Building Planets

Naturally, the next thing to do was to build a planet so that these realistic landscapes would have a geometrically correct context in which to reside. Fortunately, planets, being covered with terrains that are fractal, are themselves fractal. So we just need to build bigger fractal terrains to cover a planet.

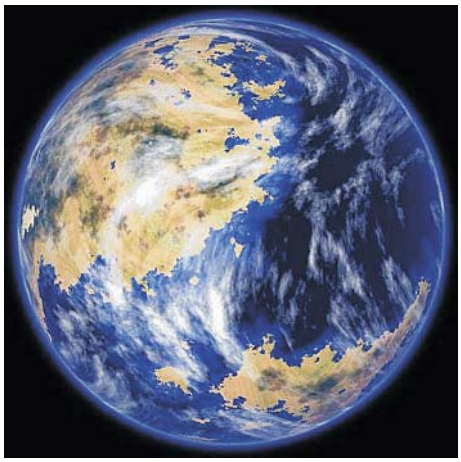


FIGURE 20.3 A preliminary Earth-like planet model.

When I set out to build my first planet, seen here in Figure 20.3, it was apparent that we needed better fractal terrains. The kind of fractals we had all been so pleased with up to that time weren't really up to the job of modeling a whole planet—they were just too monotonous. So I created some new fractals, *multifractals,* that had a little more variety, as seen in Figure 20.4. Here's the difference: See how the coastline in Figure 20.3 has pretty much the same "wiggliness" everywhere on the planet? In Figure 20.4 you can see that the coastline is pretty smooth and uncomplicated in some places and pretty "rough" or "wiggly" in other places. That's pretty much all you need to know about multifractals—no kidding! They're another step toward the true richness and complexity we see in nature. The cool thing is that they don't complicate the math much at all, which is a very good thing, if you ask me.

There are other interesting aspects to modeling a planet. One that is not so obvious is the atmosphere. Landscape painters have known for hundreds of years that the atmosphere gives the only visual indication of truly large scale in a rendering. Leonardo wrote about it in his journal. Computer graphics have used atmospheric effects for as long as we've been making realistic renderings of fractal mountains. But it turns out that, in order to get the atmospherics to work really well, even on a local scale, you can't use the simplest and easiest atmospheric model—a flatland model. You have to use an atmosphere that curves around a planet. You've seen the sun lighting clouds from underneath well after it's set—you can't get that with a flatland model! So for practical reasons of getting things just so, you end up having

Team LRN

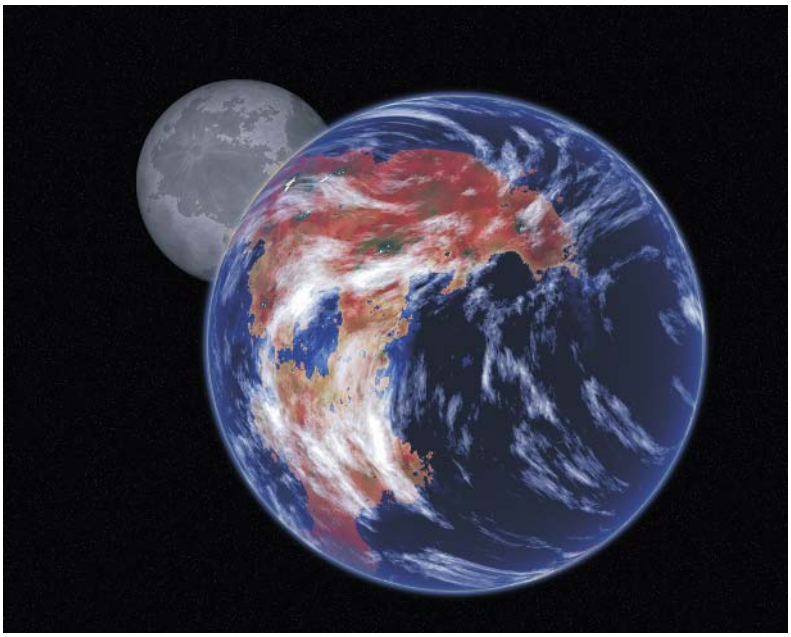


FIGURE 20.4 *Gaea & Selene* shows two procedural planet models. The clouds are the same ones seen in Figure 20.7. Note the multifractal coastline: at some places it is quite smooth, while at others it is quite convoluted, with many islands. Copyright © F. Kenton Musgrave.

to model Earth's atmosphere quite accurately, just to get sunsets to look right. Look up the word "atmosphere"—it literally means "sphere of vapor." Another indication of "global context."

Is the atmosphere fractal? Not in any obvious way, even though clouds certainly are. When I was a graduate student working under Mandelbrot, who was basically paying me to invent new fractal terrain models and make beautiful images of them, I worried that I was spending too much time on my atmosphere models. When I asked him about it, he, in true form, quipped mysteriously, "Well, many things that do not appear to be fractal are, in fact, fractal." It turns out that global circulation and the distributions of pollutants and density layers are fractal. Someday we'll have the power to model these things in MojoWorld, but not yet in this, the second year of the third millennium AD. Also, the path of photons as they are scattered by the atmosphere is fractal, not that it's of any consequence to us in making our pictures. Fractals are, indeed, everywhere.

Team LRN

Building a Virtual Universe

If landscapes need a global context, so do planets. Planets orbit suns in solar systems, stars tend to form in clusters, which in turn reside in and around galaxies, which gather in clusters and superclusters, right up the largest-scale features of our universe, which are in turn attributable to quantum fluctuations in the early universe, according to current cosmological theory. (If you want an explanation of *that,* you'd better ask Jim Bardeen, who wrote the part of MojoWorld that gives us continents with rivers and lakes. Jim is an astrophysicist who helped Stephen Hawking work out the original theory of black holes and now works on exactly that aspect of cosmology. He does rivers for us on the side, in his spare time.) Fortunately, the distribution of stars and galaxies, and the beautiful shapes of the stellar and interstellar nebulae that we're constantly getting ever better pictures of, are all quite fractal. In coming years, we here at Pandromeda have every intention of generating an entire synthetic universe that lives inside your computer. The path is clear as for how to do it. It will just take time to develop it and a lot of computer power to make it so. I, for one, am anxious for that future to arrive already!

Okay, so there's the big picture. Now how are we going to build this universe? What does it take?

Fractals. Lots of fractals.

WHAT IS A FRACTAL?

Let's get to first things first: What exactly is a fractal? Let me offer this definition:

> *fractal:* a complex object, the complexity of which arises from the repetition of a given shape at a variety of scales

Here's another definition, from *www.dictionary.com:*

> *fractal* (noun): A geometric pattern that is repeated at ever smaller scales to produce irregular shapes and surfaces that cannot be represented by classical geometry. Fractals are used especially in computer modeling of irregular patterns and structures in nature. [French from Latin *fractus,* past participle of *frangere,* to break; *see* fraction.]

It's really that simple. One of the easiest examples of a fractal is the von Koch snowflake (Figure 20.5). In the von Koch snowflake the repeated shape is an

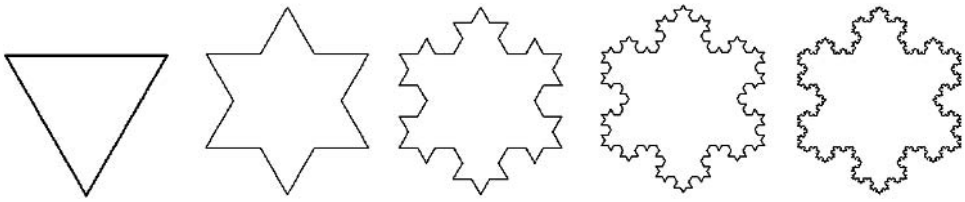


FIGURE 20.5 The von Koch snowflake: a canonical fractal.

equilateral triangle. Each time it is repeated on a smaller scale, it is reduced in size by exactly one-third. This repetition can continue at ever smaller scales ad infinitum, leading to a curve—the edge of the snowflake—that is wonderfully complex. It also exhibits some bizarre mathematical properties, but we won't go into those here.

There's lots of math we could go into about fractals, but perhaps the neatest thing about fractal geometry is that you don't need to learn any math at all to understand and use it. You can think of it as an artist, entirely in terms of visual form. Let me describe a few easy ways to think of fractals.

Self-Similarity

The repetition of form over a variety of scales is called *self-similarity*: a fractal looks similar to itself on a variety of scales. A little piece of a mountain looks a lot like a bigger piece of a mountain and vice versa. The bigger eddies in a turbulent flow look much the same as the smaller ones and vice versa (see Figure 20.6). Small rocks look the same as big rocks. In fact, in geology textbooks, you'll always see a rock hammer or a ruler in photographs of rock formations, something to give you a sense of scale in the picture. Why? Because rock formations are fractal: they have no inherent scale; you simply cannot tell how big a rock formation is unless you're told. Hence another synonym for the adjective "fractal" is "scaling": a fractal is an object that is invariant under change of scale.

Figure 20.6 shows my favorite example of a fractal in nature. Looks kind of like a puff of smoke from a smoker's mouth at arm's length, or a drop of milk in water, doesn't it? Guess how big it is. It's about 400,000 light years wide—that's roughly four thousand trillion kilometers, or 24 hundred trillion miles from one end to the other. Too big for me to imagine! In the 1970s, when I first saw this picture, taken at radio wavelengths by the Very Large Array radio telescope, I considered it the most mind-blowing image I'd ever seen. I am not sure if Benoit had even coined the term

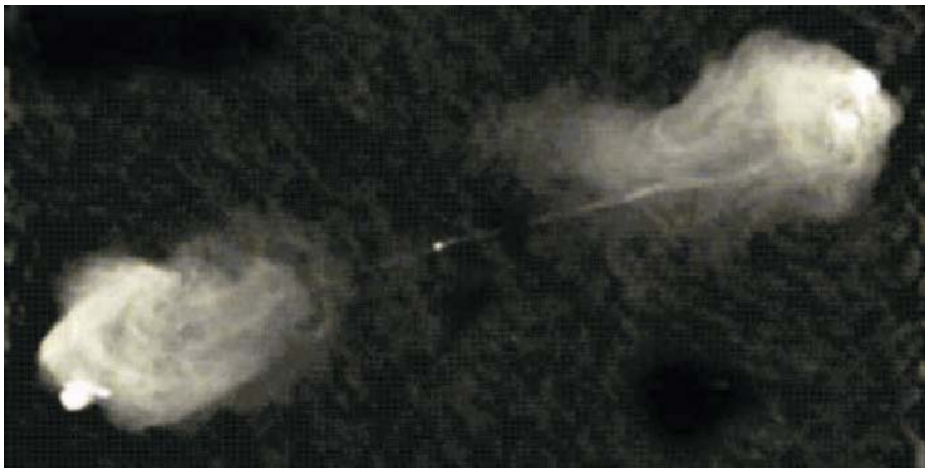


FIGURE 20.6 A fractal on a truly grand scale: jets of gas from an active galactic nucleus. VLA radio image of Cygnus A at 6 cm courtesy of NRAO.

"fractal" yet; if he had, I certainly hadn't heard it yet. But I found it stunning that this incomprehensibly large object looked so ordinary, even small. That's a fractal for you. You can recognize one immediately, even if you don't know it's called a fractal.

Dilation Symmetry

My favorite, easy way to grasp the idea of "fractal" is as a new form of symmetry: *dilation symmetry*. You're probably already familiar with symmetries such as the mirror symmetry by which the human body is pretty much the same on both sides when mirrored across a line down the middle, and perhaps the rotational symmetry whereby a square remains unchanged by a rotation of 90°. Dilation symmetry is when an object is unchanged by zooming in and out. Turbulence is like that; hence we can't tell how big that turbulent puff of gas in Figure 20.6 is until we're told.

Imagine, if you will for a moment, a tree branch. A smaller branch looks pretty much like a larger branch, right down to the level of the twigs. This is dilation symmetry. The same goes for river networks: smaller tributaries and their networks look much like the larger river networks. Figure 20.7 shows this in some of Jim Bardeen's

Team LRN

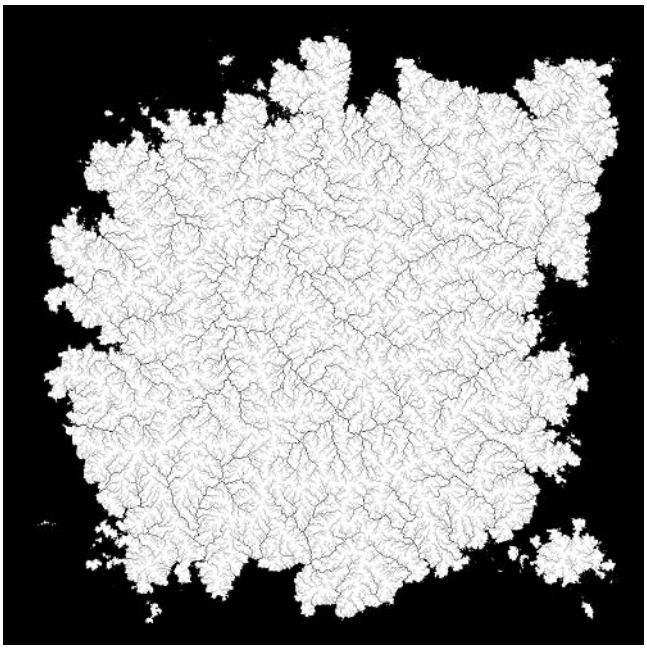


FIGURE 20.7 A fractal river drainage network for a MojoWorld continent.

river networks on a MojoWorld continent. Clouds, mountains, coastlines, and lightning are like that, too; smaller parts look just like larger parts. There is a catch: unlike the Koch snowflake, they aren't *exactly* the same at different scales, only qualitatively so. This leads to our next distinction in fractals: *random fractals*.

Random Fractals

Random fractals may be loosely defined as *fractals that incorporate random variables in their construction*. The random variable may be some quantum process, like the probability of a given air molecule scattering a passing photon, or a pseudo-random variable in a computer program, as we might use to determine the altitude of a point on a fractal terrain. Computers are always deterministic, so we don't have truly random variables in computer programs, only ones that are designed to look random while being, in fact, deterministic. "Deterministic" means that a given input always generates the same output. This determinism is a good thing: it is why we always get the same MojoWorld from a given scene file, even though what we find

Team LRN

there is unpredictable. If the computer were producing truly random variables, we might get slightly better MojoWorlds (for very obscure mathematical reasons), but we wouldn't be able to roam around and come back to the same place again.

The point is that self-similarity comes in at least two flavors: *exact* self-similarity, as in the Koch snowflake where every part is exactly the same as every other if you rotate it properly, and *statistical* self-similarity, as in all the natural phenomena I've mentioned. In nature, self-similarity is usually of the statistical sort, where the statistics of random behaviors don't change with scale. But you needn't worry any about statistics—to the human eye these fractals look similar at different scales, no doubt about it, without any reference to numbers, statistics, or any other fancy mathematics.

A BIT OF HISTORY OF FRACTAL TERRAINS

Like all intellectual revolutions, fractal geometry did not happen overnight. Rather, it was an extension of the work and observations of many people working in many different fields. But let me perform the standard practice of historians and make a complex story simple.

The Mathematics

Fractals were noticed by mathematicians around the turn of the 20th century. They noted their mathematically bizarre behavior, labeled them "monsters," and left them for posterity.

Benoit Mandelbrot had an uncle who was a famous mathematician. He assured the young Benoit that the person who cracked this mathematical case could make a name for himself. Benoit was not immediately interested, but the ideas festered (my word, not his!) in his mind, and he eventually came to work on such things as a researcher at IBM. In 1982 he published his classic book *The Fractal Geometry of Nature,* which introduced the world to fractals. In 1987 I was fortunate to be hired as Benoit's programmer in the Yale math department. I was to work on fractal terrain models that included river networks, a research project that didn't pan out for us. In 1988 *The Science of Fractal Images* was published, edited by Heinz-Otto Peitgen and Dietmar Saupe, whom I had gotten to know at UC Santa Cruz in 1986. In it Mandelbrot issued a challenge to the world to solve the difficult problem of creating fractal terrains with river networks. In 1989 Craig Kolb, Rob Mace, and I published a paper titled "The Synthesis and Rendering of Eroded Fractal Terrains," which described a way to create rivers in fractal terrains, although that method

Team LRN

remains mathematically and computationally intractable to this day. A little later Jim Bardeen solved the problem in the way that Benoit had in mind; his latest solution appears in MojoWorld. In 1993 I completed my doctoral dissertation "Methods for Realistic Landscape Imaging," which was pretty much a compilation of the various papers and course notes that I had published over the last six years on various aspects of modeling and rendering realistic forgeries of nature. In it I described my multifractal terrain models.

That's a very brief sketch of the academic mathematical history of fractal terrains. Now for the mathematical imaging track.

Mathematical Imaging of Fractal Terrains

Mandelbrot divides the history of computer images of fractal terrains into three eras: the Heroic, the Classical, and the Romantic. The Heroic Era is characterized by the work of Sig Handelman, who made the first wire frame renderings of Benoit's terrain models. According to Benoit, it was a heroic effort in the 1970s to get even a wire frame image out of the computer, hence the name of that era. Handelman's images are mostly lost in the mists of time, alas, although one or two appear in *The Fractal Geometry of Nature* (Mandelbrot 1982). Next came the work of Richard Voss, who made the first realistic (for that time, at least) images, such as the classic *Fractal Planetrise,* which graces the back cover of *The Fractal Geometry of Nature*. Voss's work comprises the Classical Era. Richard fleshed out the mathematics and rendering algorithms required to make beautiful and convincing forgeries of nature. Next came my work, in which I brought various artistic subtleties to the forefront. As I went on at length about artistic self-expression, Benoit calls my work the Romantic Era. Benoit has generously credited me with being "the first true fractal-based artist." Figures 20.8–20.10 are a few of my personal favorites from my own body of work of that era. You may notice that I was fascinated with planets right from the start.

The Computer Graphics Research Community

Then there's the computer graphics track. In 1979 Loren Carpenter, then at Boeing, made the groundbreaking computer animation *Vol Libre,* the first animated flyby of a synthetic fractal terrain. In 1982 Loren, now senior scientist at Pixar and a member of Pandromeda's distinguished board of advisors, published with Alain Fournier and Donald Fussell the paper "Computer Rendering of Stochastic Models," which introduced the commonly used polygon subdivision method for generating fractal

FIGURE 20.8 *Blessed State* is also a polygon subdivision terrain—compare it to the terrain models seen in Figures 15.7 and 20.9. The water is the "ripples" texture. The moon is a very simple fBm bump map applied to a white sphere; a more realistic moon model would be too busy for the visual composition. Copyright © F. Kenton Musgrave.

FIGURE 20.9 *Zabriskie Point* illustrates distorted fBm clouds and the rounded character of terrains constructed from the Perlin *noise* function. A vector-valued fBm function, with the vector interpreted as an RGB color value, has been used to perturb the color of the terrain. Such coloring is also seen in Figure 17.11. It is a prototype of the more sophisticated coloring seen in Figures 15.15, 16.6, 20.18, and 20.20. Copyright © F. Kenton Musgrave.

Team LRN

FIGURE 20.10 *Pleiades* has all the major elements of a procedural fractal universe. Copyright © 1996 F. Kenton Musgrave.

terrains. That precipitated a bit of a feud with Mandelbrot, but it was before my time in the field, so I'll say no more about that. Also in 1982, Loren and the rest of the distinguished crew at Pixar showed us the first MojoWorld (if I may be so presumptuous) on the big screen in the "Genesis Sequence" in *Star Trek II: The Wrath of Kahn*. That blew our minds. In 1985 the late Alain Fournier came to UC Santa Cruz to teach a course, "Modeling Natural Phenomena," that changed my life and set the course of my career.

Alain mentored me in the area and advised me in my master's research. Later in 1985 Ken Perlin and Darwyn Peachey published twin papers that introduced the procedural methods that have pretty much driven my entire career. Thanks, guys! They had some really cool pictures in those papers; I saw a world of possibility (so to speak) in their methods, and the rest is, well, history. In 1986 Dietmar Saupe and Heinz-Otto Peitgen came to the UC Santa Cruz math department, and I took Dietmar's course "Fractals in Computer Graphics." In 1987 Dietmar recommended me to Mandelbrot for the job at Yale. (I always tell my students: "There's no substitute for dumb luck.") In 1993 I finally graduated from Yale with a terminal degree—

Team LRN

a Ph.D., but I prefer that other term for it—at the ripe old age of 37. Can you say "professional student?" In 1994 Matt Pharr, Rob Cook, and I created the *Gaea Zoom* computer animation (see Figure 16.3 and *www.kenmusgrave.com /animations.html*). I believe it was the first MojoWorld with adaptive level of detail, that is, a synthetic fractal planet that you could zoom in and out from, without nasty artifacts described by obscure mathematics that I won't go into here. Suffice it to say, it's not so easy to make a planet that looks good from both near and far, doesn't overload your computer's memory, and renders in a reasonable amount of time. The *Gaea Zoom* took two weeks to render on four supercomputers, so we weren't quite there yet in 1994. Finally, in 2001, we were. Hence MojoWorld launched that year—it really couldn't have been launched even a year earlier.

The Literature

And then there's the track of explanations of fractal terrains. First came the technical papers that even *I* never could really understand. Then came *The Science of Fractal Images* in 1988 (which I even wrote a tiny part of), in which Richard Voss and Dietmar Saupe cover everything you'd ever want to know about the mathematics of these and other kinds of fractals. Next came our book, *Texturing & Modeling: A Procedural Approach*, first edition in 1994, second in 1998, third edition circa 2003 in your hands now, by David Ebert, Darwyn Peachey, Ken Perlin, Steve Worley, and me. In it I explain how to build fractal terrains from a programming perspective. Still pretty technical, but the standard reference on how to program fractal terrains and even entire MojoWorlds. And now there's this little exposition, in which I'm trying to explain it all to the nontechnical reader.

The Software

Last but not least, there's the software track of the history of fractal terrains. For some time there existed only the various experimental programs created by us academics. They couldn't be used by the average person or on the average computer. The first commercial software that included fractal terrains was high-end stuff like Alias. I'm afraid I must plead ignorance of those high-end packages. I was just a lowly graduate student at the time and, while I had access to some mighty fancy computers to run my own programs on, we certainly couldn't afford such top-of-the-line commercial software, and they weren't giving it away to university types like us despite our constant and pathetic pleas of poverty and need.

Team LRN

The first affordable commercial fractal terrain program that came to my attention was Vistapro, around 1992. With its flight simulator interface for making animations, it was really cool. You can still buy Vistapro, although it's a bit quaint by today's technological standards. Next, I believe, came Bryce 1.0, in 1994, written primarily by Eric Wenger and Kai Krause for what was then HSC Software, then MetaTools, then MetaCreations, now defunct. (Corel now owns the Bryce name and is carrying the product forward.) Bryce 1.0 was Macintosh-only software and mighty cool. It put a user-friendly interface on all the procedural methods that my colleagues and I had been going on about in the academic literature for years, and made it all accessible to the average home computer user. Since then, Animatek's World Builder and 3D Nature's World Construction Set have released powerful, semi-high-end products priced around $1000. Natural Graphics' Natural Scene Designer, E-on Software's Vue d'Esprit, and Matt Fairclough's Terragen shareware program have filled out the low end at $200 and less. Each product has its specialty; each can create and render fractal terrains. Meanwhile, Bryce is up to version 5.0. I personally worked on Bryce 4.0 for MetaCreations until December 1999, when MetaCreations imploded. The very next day FractalWorlds, now Pandromeda, was launched to make MojoWorld a reality. We're the first to come to market with entire planets with level of detail, thus opening the door to cyberspace.

Disclaimers and Apologies

So there's Doc Mojo's *Close Cover Before Striking History of Fractal Terrains*. Sure, it's biased. I worked for Mandelbrot. I come from the academic side and was camped more with the mathematicians than with my real colleagues, the computer science/computer graphics people. I've left out a lot of important contributions by friends and colleagues like Gavin Miller, Jim Kajiya, and many, many more. I'm keeping it brief here; if you want the exhaustive listing of who's done what in the field, see the bibliography of this book or my dissertation. And I must state that, although many people think so, I am *not* a mathematician. Believe me, having a faculty office in the Yale math department for six years drove that point home! My degrees are in computer science. But I'm really a computer artist more than a computer scientist. My contribution has been mostly to the artistic methods, ways to make our images of fractal terrains more beautiful and realistic. I'm a mathematical lightweight; I just do what I have to do to get what I want. And all my reasoning is visual: I think in terms of shape and proportion, even if I do translate it into math in order to make my pictures. To me all the equations just provide shapes and ways to combine them.

Team LRN

The Present and Future

The abstract of my 1993 doctoral dissertation ends with this sentence:

> Procedural textures are developed as models of mountains and clouds, culminating in a procedural model of an Earth-like planet that in the future may be explored interactively in a virtual reality setting.

The planet model I was referring to is Gaea, seen in Figures 20.4 and 16.3, implemented as a C language version of the "terran" texture presented in Chapter 15. In MojoWorld we finally have the interactive planet I envisioned and an endless variety of others as well. It's still not quite what I'd call "virtual reality"—the real-time part is just not that realistic. Yet. But getting there is just a matter of a lot of hardware and software engineering.

Gertrude Stein once said of Oakland, California, "There's no 'there' there." If that's true for Oakland, I say it's even more true of all implementations of virtual reality up to now. MojoWorld, at last, puts the "there" there. This is the main thing that's fundamentally new about MojoWorld.

All other 3D graphics programs build the equivalent of stage sets. Stage sets are designed to be viewed from a distance and at a given resolution, whether it's that of a TV camera, a movie camera, or the human eye. If you get too close, the flaws show. (One problem TV studios face today is that the transition to high-definition, or HD, video requires that they replace their studio sets. The ones that look fine with older video technology won't cut it with HD video: all the defects in the sets show—things like dents and dings, coffee rings, and poor craftsmanship that are invisible at NTSC resolution suddenly are clearly visible. I discovered the same thing early in my career as a computer artist: images that looked great at screen resolution didn't always look so good when rendered at poster resolution.) Also, a set is always *local:* it is of finite size, and if you get too far away, it ends. And generally, if you view it from the wrong angle, the effect it's designed to create breaks down. MojoWorld isn't like that. You can go anywhere. In MojoWorld, as Buckaroo Bonzai said, "Everywhere you go, there you are." You never run out of detail, and there's always someplace new and interesting to visit.

I think of MojoWorld as a window on a parallel universe, a universe that already exists, always has and always will exist, in the timeless truth of mathematical logic. All we've done is create the machinery that reveals it and the beauty to be found there. It's been my great privilege to play a small part in the discovery/creation

Team LRN

FIGURE 20.11 *Southern Parfait Flood* is another scene from the parallel universe we call the "Mojoverse." There are countless other such scenes of beauty and fascination waiting to be discovered there. Copyright © 2002 Armands Auseklis.

of this possibility. It's been my vision for years now to make this experience accessible to everyone. With MojoWorld, we're on our way, and I, for one, am very excited about that. I hope you enjoy it half as much as I will!

Now let's get on to explaining how we build a MojoWorld so that you understand the controls that you'll be using.

BUILDING RANDOM FRACTALS

The construction of fractal terrains is remarkably simple: it is an iterative loop involving only four important factors, one of which is generally a nonissue. First, we have the *basis function,* or the shape that we build the fractal out of, by repeating it at a variety of scales. Next there's the *fractal dimension,* which controls the roughness of the fractal by simply modulating the amplitude or vertical size of the basis

function in each iteration (i.e., each time you go through the loop). Then there are the *octaves,* or the number of times that we iterate in building the fractal. Finally, we have the *lacunarity,* or the factor by which we change the frequency or horizontal size of the basis function in each iteration. Usually, we leave the lacunarity at around two and never think about it again. Let's see what the effect of each of these four factors is and how they all fit together.

The Basis Function

The basis function is perhaps the most interesting choice you get to make when building a random fractal, whether for terrain, clouds, water, nebulae, or surface textures. The shape of the basis function largely determines the visual qualities of the resulting fractal, so "choose wisely." It's fun to experiment and see the subtle and not-so-subtle visual effects of your choice of basis function. I have certainly gotten a lot of artistic mileage over the years through careful choice and modulation of basis functions. And MojoWorld has plenty of basis functions, that's for sure.

For obscure but important mathematical reasons, basis functions should (1) have shapes that are not too complicated, (2) never return values smaller than −1.0 or larger than +1.0, and (3) have an average value of 0.0. As Mick once said, "You can't always get what you want," but the basis functions in MojoWorld are designed to obey these constraints in most cases.

The main thing to keep in mind about the basis function is that its shape will show through clearly in the fractal you're building. So your choice of basis function is the most significant decision you make when building a fractal.

Fractal Dimension: "Roughness"

Fractal dimension is a powerful, if slippery, beast. I'll leave mathematical explanations to other texts. In MojoWorld we call it "Roughness." For our purposes, just think of the Roughness control as a slider that controls visual complexity. It does this by varying the jaggedness of terrain, the wiggliness of coastlines, the raggedness of clouds, and the busyness of textures. The Roughness control in MojoWorld is an extremely potent control.[1] Use it a lot! You'll find it a powerful and subtle way to

1. The way we've implemented it in MojoWorld, the Roughness control doesn't necessarily have an accurate relationship to the numerical value of the fractal dimension in the fractal you're building, but that doesn't matter—the numerical value isn't important to us in the context of MojoWorld, only the qualitative effect we're getting, which you can assess visually.

Team LRN

affect the aesthetics of your fractals. And don't be surprised if you find yourself setting its value via text entry, down to the third digit after the decimal point. It's that sensitive and powerful. Play with it and see.

Larger values make for rougher, busier, more detailed fractals (see Figure 20.12). They tend to get visually "noisy" at values over about 0.5. I generally prefer to use smaller values than most people, but, hey, it's strictly a matter of taste.

Octaves: Limits to Detail

We call the number of times we iterate, adding in more detail, the *octaves*. Fractals can have potentially unlimited detail. But that detail has to be built by the computer, so it must have limits if you want your computation to finish. In nature, fractals are always *band-limited:* there is a scale above which the fractal behavior vanishes (this is even true of the largest structures in the universe) and a scale below which it also goes away (as when we get to the scale of quantum physics). For mathematical reasons, MojoWorld has to be in control of the number of times each sample of a fractal goes through the construction loop. The explanation for why this is so is beyond the scope of this presentation. (See Chapter 17 for details on this.) Suffice it to say that controlling the number of times we go through the loop controls the amount of detail, and the amount of detail required at a given point in the image depends on its distance from the camera, the screen resolution, the field of view, and other, more subtle factors as well. Furthermore, too much detail not only wastes computation time, it also causes *aliasing*—nasty visual artifacts that we go to great lengths to eliminate in MojoWorld. So, bottom line, MojoWorld has to control the number of octaves in the fractals. That's just the way it is.

We can, however, play games with the octaves. The Detail control can reduce or increase the number of octaves, and hence the fine detail, in MojoWorld fractals. Its effects can look pretty strange in animations and when you change the rendering resolution, but it can keep your fractals from being annoyingly visually "busy"

FIGURE 20.12 Planets with fractal roughness of −0.5, 0.0, 0.5, 1.0, and 1.5.

Team LRN

everywhere, all the time. The Largest Feature Size and Smallest Feature Size controls set the band limits to the fractal. You'll get no more fractal detail above and below these scales. You have to set the Largest Feature Size to something reasonable. Keep in mind that it shouldn't be any larger than the planet, or you're just wasting computation time. The Smallest Feature Size can be left at zero. MojoWorld will eventually decide that "enough is enough" and stop generating more detail, but you'll probably get tired of zooming in long before that.

Tip: *If you build a terrain or texture that aliases in the high-quality MojoWorld renderings, use the Detail control to reduce the number of octaves until the aliasing goes away at the quality level you're using.*

Note: *The MojoWorld RTR (real-time renderer) uses a very different sampling method than the MojoWorld photorealistic renderer. It will usually suffer significant to severe aliasing when viewing a planet from a distance, whereas the photorealistic renderer will not.*

Lacunarity: The Gap between Successive Frequencies

This one is usually a nonissue, but we've made it an input parameter in MojoWorld just to be thorough, as you can get certain unique artistic effects using lacunarity. When going through the iterative loop that builds the fractal, the frequency or lateral scale of features must change at each iteration because that's how we get features at a variety of scales. The *lacunarity* determines how much the scale is changed at each iteration. Since "scale" is in this case synonymous with "spatial frequency" (of the features in the basis function), it's easiest to think of the lacunarity as the gap between successive spatial frequencies in the construction of the fractal. Indeed, "lacuna" is Latin for "gap."

Usually, we double the frequency at each iteration, corresponding to a lacunarity of 2.0. Musically, this corresponds to raising the frequency by one octave, hence the term "octaves" for the number of scales at which we create detail. Why the value of 2.0 and not something else? Well, it has to be bigger than 1.0, or you go nowhere or even backward. On the other hand, the bigger you make it, the faster you can cover a given range of scales because you're taking a bigger step at each iteration. Each iteration takes time, and when you're building a planet, you have a big range of scales to cover. So a clever person might think, "Well, then, just crank up the lacunarity!" Not so fast, Bucko. It turns out that for lacunarity values much over 2.0, you start to see the individual frequencies of the basis function. It just looks bad, as Figure 20.13

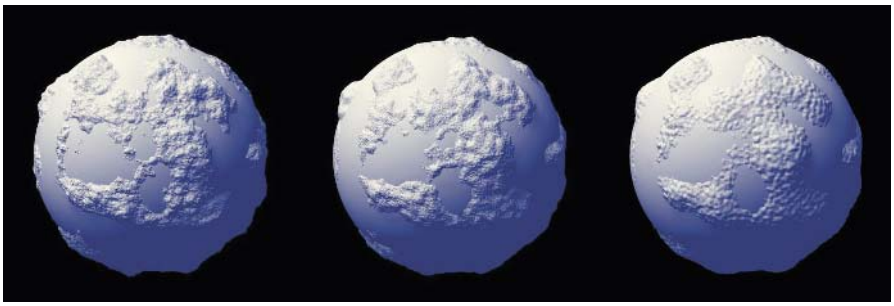


FIGURE 20.13 Planets made with a Perlin basis and lacunarities of 2.7, 5.0, and 10.0.

FIGURE 20.14 Planets made with a Perlin basis and lacunarities of 1.5, 1.9, and 2.7.

demonstrates. Similarly, Figure 20.14 shows that lacunarities less than 2.5 pretty much look alike, although close inspection will reveal artifacts at 2.7. We've gone with a default lacunarity just over 2.2 in MojoWorld, to eek out a little more speed. If you want images that are as good as they can be, I'd recommend a value more like 1.9.[2] My best advice: Don't mess with lacunarity until you really know what you're doing.

2. For obscure technical reasons, it's best not to use a lacunarity of exactly 2.0, but something close to it, like 1.9 or 2.1. Transcendental numbers are best. MojoWorld's default is the natural logarithm *e* minus one-half, or 2.718 . . . −0.5 = 2.218. . . . You might try changing the 2 after the decimal point to 1. Keep in mind you should set your lacunarity permanently before you use your fractal because changing it will change all the features except those on the largest scale, and this could completely disrupt some specific feature in your MojoWorld that you've become interested in.

ADVANCED TOPICS

That covers the fundamentals of where MojoWorld comes from and how it works. Now we'll go into some of the more esoteric aspects of MojoWorld.

Dimensions: Domain and Range

The various functions used to create textures and geometry (for example, mountains) in MojoWorld are implemented in several dimensions. What does this mean? Pay close attention, as this can be a little confusing and even counterintuitive. A *function* is an entity that maps some input, called the *domain* of the function, to some arbitrary output, called the *range*. Inside the MojoWorld program, the domain and range are all a bunch of numbers. As users, we usually think of the output as color, the height of mountains, the density of the atmosphere, and other such things. We also think of the input as things like position in space, altitude, and color, too, or as numbers like the ones we can type into the UI.

Both the domain and range of functions have *dimensions*. One dimension corresponds to a line, two to a plane, three to space, and four to space plus time. When you're creating a new function in MojoWorld, you'll sometimes have to choose the dimensionality of the domain of the function. This seems a little backward, as what you're really interested in is the dimensionality of the output, or the range of the function. Here's the catch: you can't have meaningful output of dimensionality greater than that of the input. There's just no way to make up the difference in information content.

Usually, we're working in three dimensions in MojoWorld, so that's the correct default choice to make when you're confronted with this decision. But, in general, every added dimension doubles the time required to compute the function. So you want to use as few dimensions as you can get away with. You might also want to do some special effects using lower dimensions, like determining the climate zones of your planet (implemented as a three-dimensional texture) by latitude (a one-dimensional variable). Figure 20.15 illustrates MojoWorlds made from the same function, with domains of one, two, and three dimensions.

MojoWorld also has a full complement of functions with four-dimensional domains "under the hood." These will be useful for animating three-dimensional models over time to simulate things like continental drift and billowing volumetric clouds. The user interface for animation is a complicated thing, however, so we decided to leave it for a future version of MojoWorld, when we've had time to do it right.

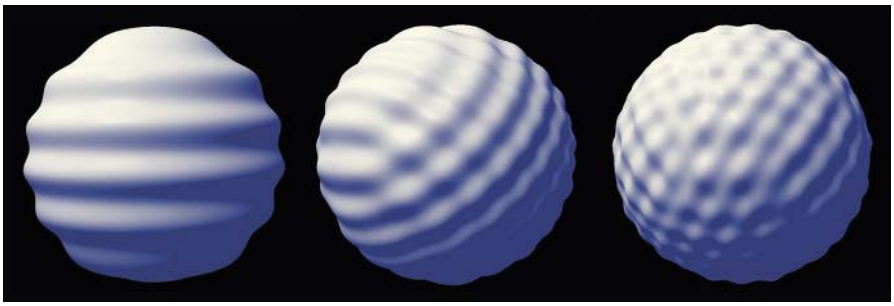


FIGURE 20.15 Planet made from a sine function with one-, two-, and three-dimensional domains.

Hyperspace

You may have noticed that we go on about *hyperspace* a lot in our MojoWorld propaganda. Hyperspace is whenever you go up one dimension: a plane is a hyperspace to a line, and three dimensions is a hyperspace to a plane. Add time to space, and you have a hyperspace to the three dimensions we're most familiar with. Well, it's really easy to keep adding more dimensions. Take the three dimensions of space and add a color, for example. Because the human eye has three different color receptors in the retina, one each for red, green, and blue light, human vision has three color dimensions—hence the RGB color space used in computer graphics. (Some birds have six; they live in a world with far richer color than we monkeys.) It takes three values to specify a color for the human eye: one each for red, green, and blue. Each is independent—part of the definition of a *dimension*. From *www.dictionary.com:*

> **dimension:** . . . 4. *Mathematics*. a. One of the least number of independent coordinates required to specify a point in space or in space and time.

In our example we have three dimensions for space and three for color, for a total of six dimensions. Presto—a hyperspace! Not hard to do at all, eh?

Now think for a minute of all the values you may assign to make a MojoWorld—things like planet radius, sea level, color, atmospheric density, fractal dimension, and so on. For an interesting planet, there'll be hundreds, even thousands of variables involved. From a mathematical standpoint, each independent variable adds another dimension to the space that the planet resides in. The more complicated the MojoWorld, the higher the dimensionality of the space it resides in. Each

lower-dimensional space, as for the same MojoWorld with one less color specified, is called a *subspace* in mathematics. And, of course, each higher-dimensional space is a *hyperspace*. There's no limit to the amount of complication you can add to a MojoWorld, and so there's no limit to the dimensionality of the master MojoWorld hyperspace. Pretty mind-bending, ain't it? I certainly think so!

The variables used to specify a MojoWorld are called *parameters*. So the master hyperspace spanned by the possible parameters is rightly called *Parametric Hyperspace*. (We took out a trademark on the name because that's what corporations do.) It's simply the most succinct and accurate way to think and talk about how MojoWorld works. A pure MojoWorld scene file, uncomplicated by content such as plants, cities, monkeys, and the like, needs only encode the numbers that specify the parameter settings. Everything else is generated at run time from these values. This set of parameter values specifies the point in Parametric Hyperspace where the MojoWorld resides. Load them into MojoWorld, and MojoWorld "beams" you there—hence we call them *transporter coordinates*. Very sci-fi, but very scientifically and mathematically accurate. MojoWorld Transporter let's you beam yourself to these places and explore them in three dimensions. If you want access to all of

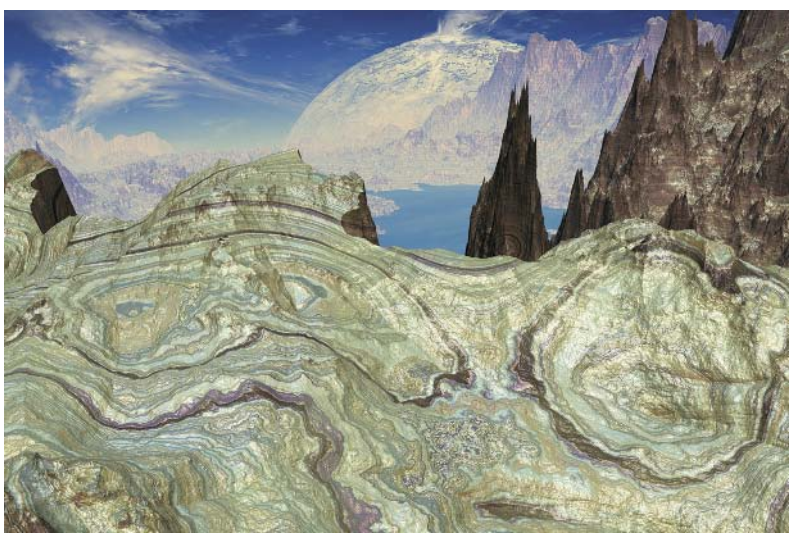


FIGURE 20.16 *At the Edge* illustrates the kind of beautiful surface textures that can be built using the procedural texture engine that is the heart of MojoWorld. Copyright © 2002 Allison Morris.

Parametric Hyperspace, however, and have the ability to mess with all of the parameters, you need MojoWorld Generator.

The Basis Functions

MojoWorld has by far the richest set of basis functions ever seen in a random fractal engine, so don't be surprised to find the choices a bit bewildering at first. They're all based on methods from the academic literature in computer graphics. If you're an advanced graduate student in the field, they should all be familiar. If you're not, don't worry—not everyone needs a Ph.D. in computer graphics! (Very few do, indeed.) You can familiarize yourself with the choices by least two routes: you can plunge right into the lists of basis functions and their controlling parameters in the MojoWorld Texture Editor, or, if you're less patient and intrepid (like myself), you can keep examining the way various MojoWorlds that you like have been built and note the basis functions used in fractals that you particularly like. The variety of basis functions available in MojoWorld far surpasses anything I, personally, have ever had before, and it will be a long time—probably *never*—before I get around to trying every basis function that's possible in there. In fact, once you get to distorting or applying curves to the basis functions, the possibilities are basically infinite, so no one will ever try them all.

Because there are so many, I'm not going to try to describe all of MojoWorld's basis functions here. Rather, I'll describe the basic classes into which they fall and the fundamental visual qualities of each.

Perlin

The best and fastest basis function, in general, is a Perlin *noise*. Ken Perlin introduced this famous basis function in his classic 1985 SIGGRAPH paper "An Image Synthesizer," and it's the basis function that launched a thousand pictures (and my own career). Ken—and everybody else—calls it a "noise function." In the context of fractal mathematics this term is rather misleading, so in MojoWorld we call Perlin *noise* a "Perlin basis." At any rate, there are several flavors of the Perlin basis, and we have them all in MojoWorld, plus a new one.

The Perlin basis consists of nice, smooth, random lumps of a very limited range of sizes. Its output values range between −1.0 and 1.0. It's ideal for building smooth, rounded fractals, such as ancient, heavily eroded terrains as seen in Figures 16.2 and 18.2. One of everyone's favorite terrain models is the so-called "ridged multifractal" seen in Figure 20.18. It looks completely unlike an ordinary Perlin fractal but is

Team LRN

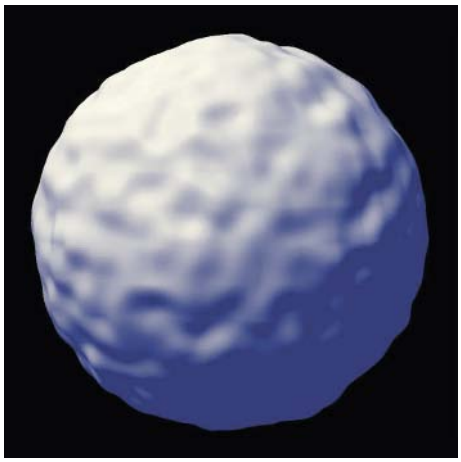


FIGURE 20.17 A planet with relief from a Perlin basis.

closely related: it's made by taking the absolute value of the Perlin basis—that is, by changing the sign of all negative values to positive—and turning that upside down. Figure 20.19 shows visually how this process works. The result is a basis that has sharp ridges. You can use it in a monofractal function to get terrain, as seen in Figure 20.20, or in a multifractal to get terrain, as in Figure 20.18.

Voronoi

The Voronoi, or "cellular," basis functions are cool and useful, but slow. Steve Worley introduced the Voronoi basis in his 1996 SIGGRAPH paper (Worley 1996). It has a cell-like character, kind of like mud cracks with pits drilled into the middle of each tile of the mud (see Figure 20.21.) The pits are conical when you use the "distance" contour and rounded (actually, parabolic) when you use the "distance squared" contour. The mud tiles are flat plateaus of random height when you use the "cell ID" contour. The value of the Voronoi basis at a given sample point derives from the distance of that sample point from a random point in space, called the "seed," and one or more of its neighbors in a stored set of seeds. There is a ridge at the perpendicular bisector of the line between the random point and the chosen neighbor. That neighbor can be the first, second, third, or fourth closest neighbor to the point. You don't have to worry about which number you choose; rather, just look at the quality of the resulting texture and choose one you like. You can also

Team LRN

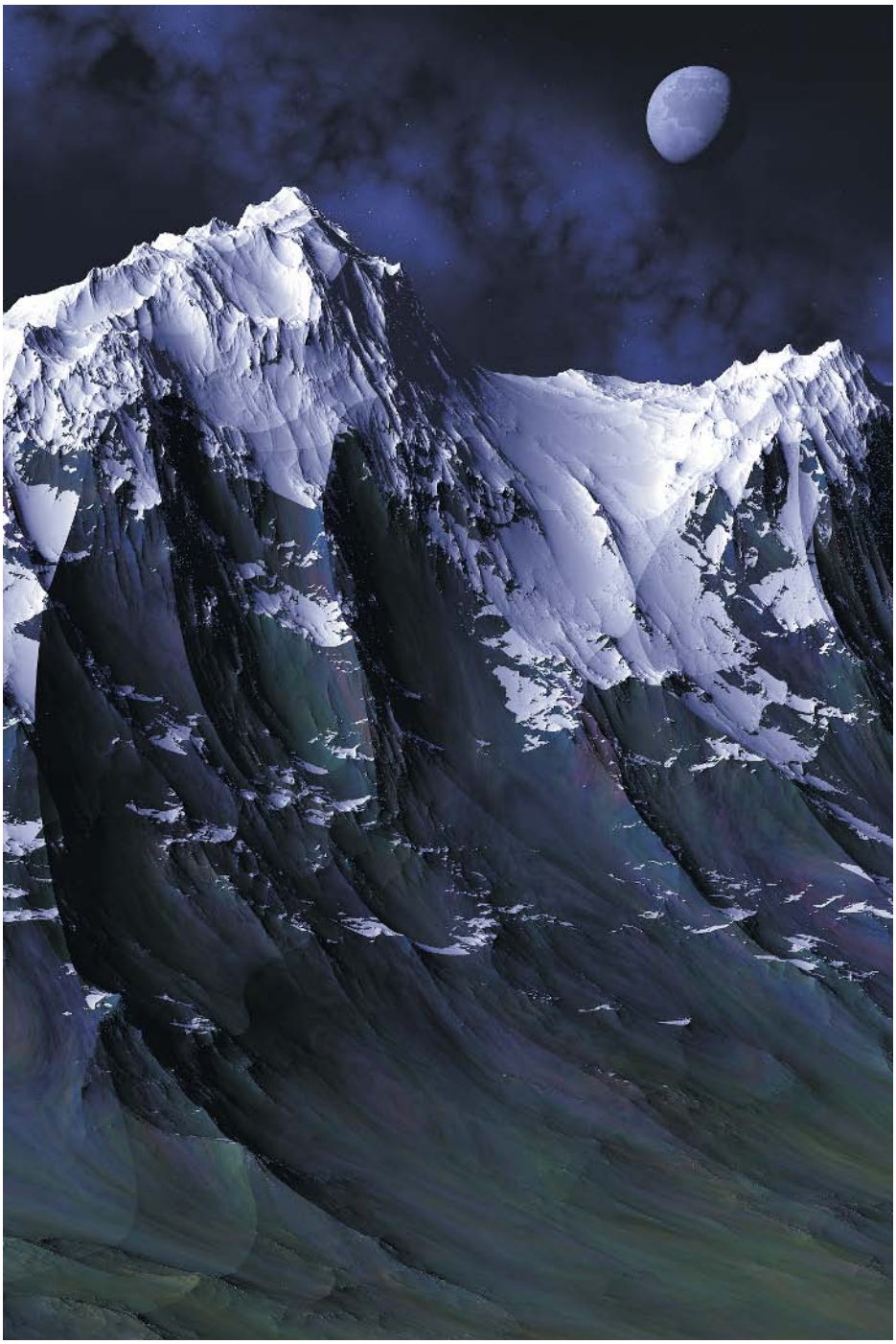


FIGURE 20.18 *Emil* is a terrain made from a ridged multifractal, a variety of Perlin fractal. Copyright © 1996 F. Kenton Musgrave.

Team LRN

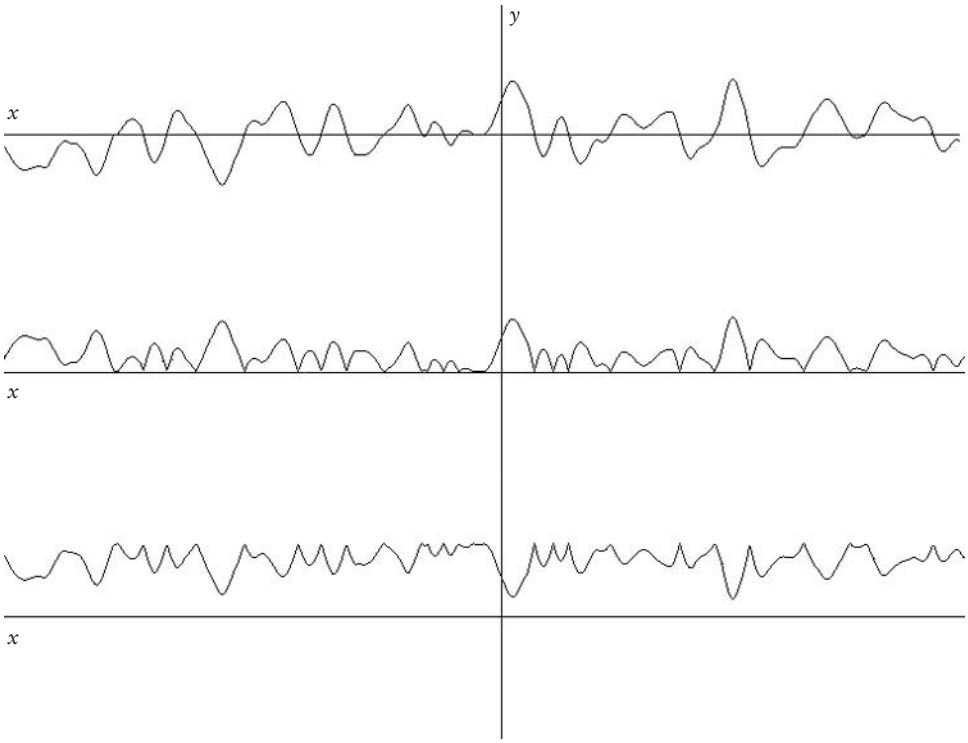


FIGURE 20.19 Building a ridged basis function from a Perlin basis: the Perlin basis, its absolute value, and one minus the absolute value.

choose the differences between the first and second, second and third, or third and fourth neighbors. Again, figure out what that means only if you want to; otherwise, just examine the quality of the texture you get and choose one you like for aesthetic reasons.

Voronoi basis functions have ridges like the ridged Perlin basis, only they're all straight lines. Usually, you'll want to apply a fractal domain distortion to Voronoi fractals, to make those straight lines more natural (read: wiggly).

Sparse Convolution

In 1989 John Lewis introduced the sparse convolution basis: the slowest, technically "best," and most flexible of all (Lewis 1989). I'd say its time has not yet come—we simply need faster computers before this basis is going to be practical. But I'm

Team LRN

FIGURE 20.20 *Slickrock* is a monofractal built from a ridged Perlin basis. Copyright © 1993 F. Kenton Musgrave.

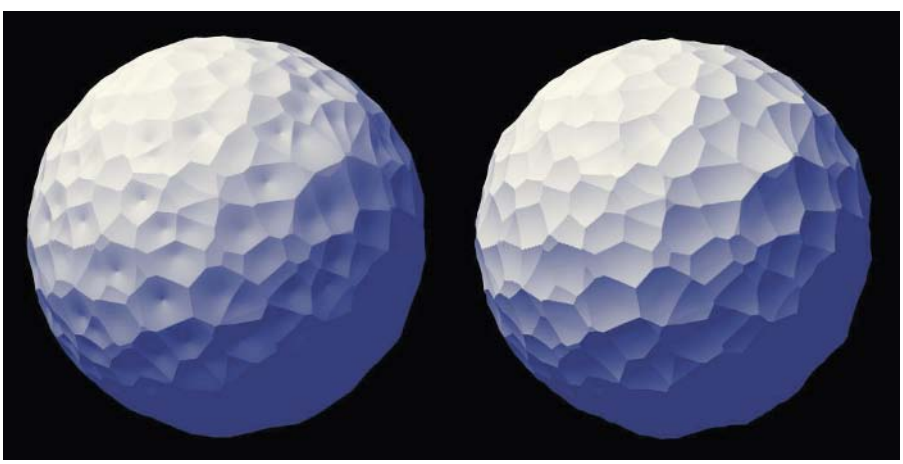


FIGURE 20.21 Planets made from the Voronoi basis, first neighbor. "Distance" Voronoi on the left, "distance squared" on the right. "Distance" has conical pits; "distance squared" has smooth pits.

Team LRN

personally obsessed with basis functions and I wanted it in there, so I put it in there. And to my surprise, people are using it! (See Figure 20.22.)

The sparse convolution basis generates random points in space and splats down a *convolution kernel* around those points. (That funny name comes from some more pointy-headed math terminology.) The kernel can be literally anything. In MojoWorld version 1.0 it can be a simple, radially symmetric shape that you choose from a list of options or building the curve editor. In later versions, we'll get into some bizarre and powerful kernels like bitmaps. Personally, I look forward to building planets out of Dobbsheads.

Various

Then there are the various other basis functions that I've thrown in for fun. See if you can find a creative use for them! Some of you may wonder why I haven't included some of the ones found in other texture engines such as Bryce's (which was written by Eric Wenger and myself). Well, some of those "noises" are really textures, not basis functions. You can obtain similar results, with far more flexibility, using MojoWorld's *function fractals,* which will build a fractal from whatever function you pass to them—and probably alias like crazy while they're at it. Such aliasing is why many things that are used as basis functions in Bryce shouldn't be used for that. Others, like Eric's "techno noise," don't lend themselves to the level-of-detail schemes used in MojoWorld. As a rule, for deep mathematical reasons that I

FIGURE 20.22 A planet made from the sparse convolution basis, using a cone for the kernel.

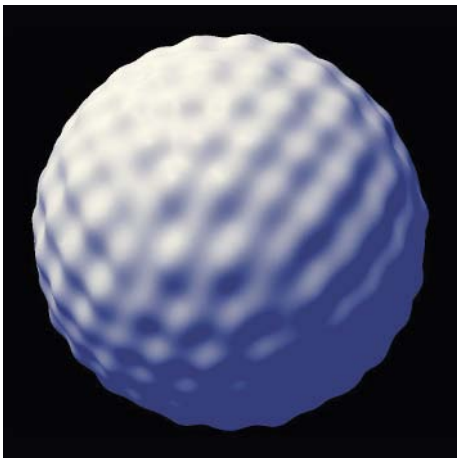


FIGURE 20.23 A planet made from a 3D sine basis.

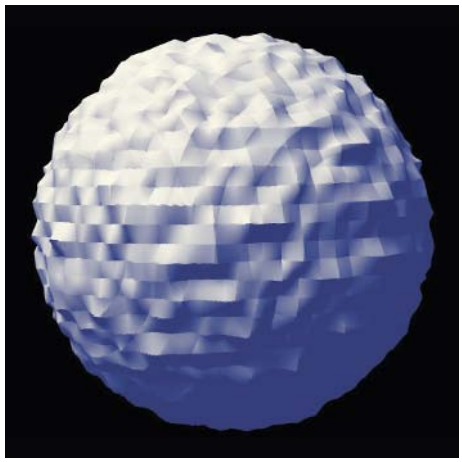


FIGURE 20.24 A planet made from the linear basis function.

won't go into here, basis functions should be visually simple. So all of MojoWorld's basis functions are—with the exception of sparse convolution when using a complex kernel. Keeping to this constraint of simplicity, here are a few more basis functions.

First there's the venerable *sine* wave. As a function of more than one variable, it's a little ambiguous what a sine wave should be. In MojoWorld we multiply sine waves along the various dimensions. As the sine function is periodic, anything built using the sine basis will be periodic (see Figure 20.23). Periodic phenomena are quite common in nature, but they tend to look unnatural in synthetic images. Nature can get away with things that we can't.

The *linear* basis function is a simpler version of the simplest Perlin basis. It uses linear interpolation of random offsets, rather than the smooth cubic spline used in the Perlin bases. It will give you straight, sharp creases (see Figure 20.24). Not very natural-looking, but I'm sure someone will find a use for it.

The *steps* basis is simpler still: it's just a bunch of random levels in a cubic lattice (see Figure 20.25). And then there's the procedural texture that any student of computer graphics programs first: the common *checkerboard*. In MojoWorld, the checkerboard basis alternates between 1.0 and −1.0. The little sawtooth artifacts you see on the cliff faces in the steps and checkerboard planets (Figures 20.25 and 20.26) are the micropolygons that compose them. They can be made smaller, at the expense of

Team LRN

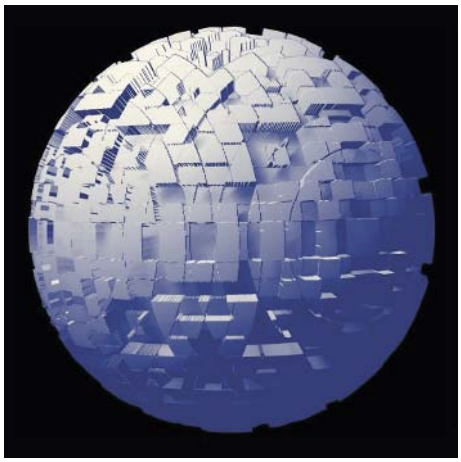


FIGURE 20.25 A planet made from the steps basis function.

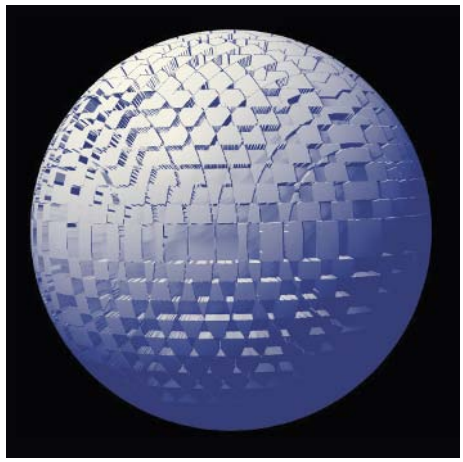


FIGURE 20.26 A planet made from the checkerboard basis function.

longer render times, but they'll never go away completely. Discontinuous basis functions like these are mathematically evil. And so they really shouldn't be used, but what the heck, MojoWorld is for play as well as serious work, so not everything has to be perfect.

The Seed Tables

Here's another MojoWorld first. Call me a noise dweeb, a basis function junkie, or just plain ill advised, but . . . "I just had to." The Voronoi and sparse convolution basis functions are built from tables of random points in space. It just turns out that there are many forms "random" can take. (Take a class in probability or statistics and learn to hate them.) So I implemented a mess of different ones and whacked 'em into MojoWorld.

Look at them closely and try to distinguish the subtle visual differences between them (see Figure 20.27). In general, the ones with larger numbers have a denser spatial distribution. The ones with "uniform distribution" are more dense in some areas and less dense in others—more heterogeneous, in a word. Uniform distribution means that each cell has an equal probability of having any of the possible range of seeds per cell in each cell. Thus the "0–2 per cell, uniform distribution" seed table has cells that have zero, one, or two seeds per cell, with equal probability of each.

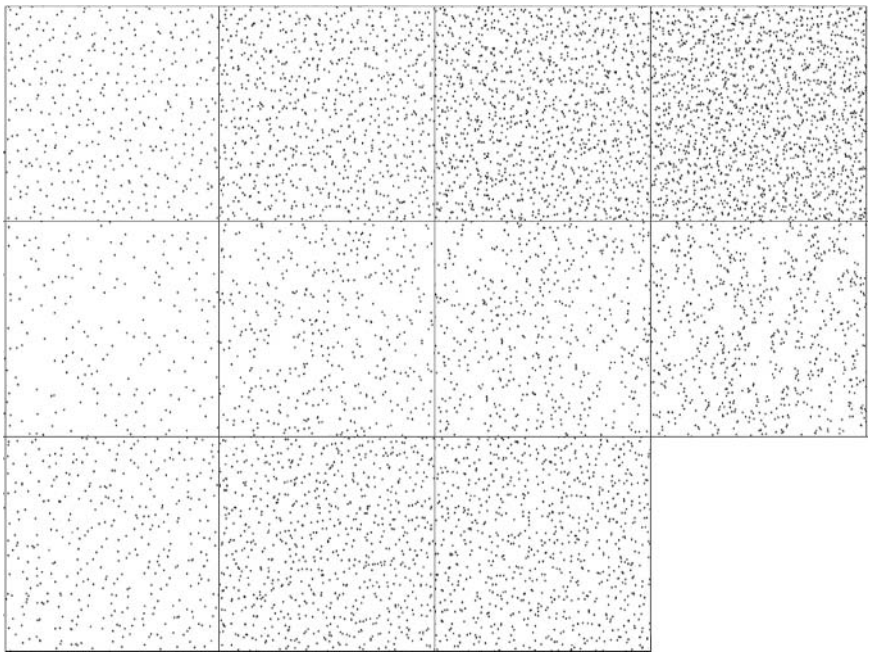


FIGURE 20.27 The distributions of seeds in the 11 seed tables available in MojoWorld: 1, 2, 3, and 4 seeds per cell; 0–1, 0–2, 0–3, and 0–4 seeds per cell, uniform distribution; and 0–2, 1–3, and 0–4 seeds per cell, Gaussian distribution.

The seed tables with "Gaussian distribution" are more subtle: there is a higher probability of a given cell having the middle of the possible number of seeds per cell than the greatest or smallest possible number. This is the good old "bell curve" that statisticians are so fond of. So the "0–4 per cell, Gaussian distribution" seed table will have mostly cells with two seeds each, very few with zero or four seeds, and an intermediate number of cells with one or three seeds.

At any rate, just evaluate the seed tables visually and use them if you like, or just ignore them—they are a *very* advanced feature for subtle effects. Figure 20.28 illustrates some of the extremes in the visual consequences available with different random seed tables. Note that even these "extremes" are only subtly different; there are other tables with intermediate values to make the possible transitions all but imperceptible. A few perfect masters of MojoWorld will someday find these subtle differences useful, I predict.

Team LRN

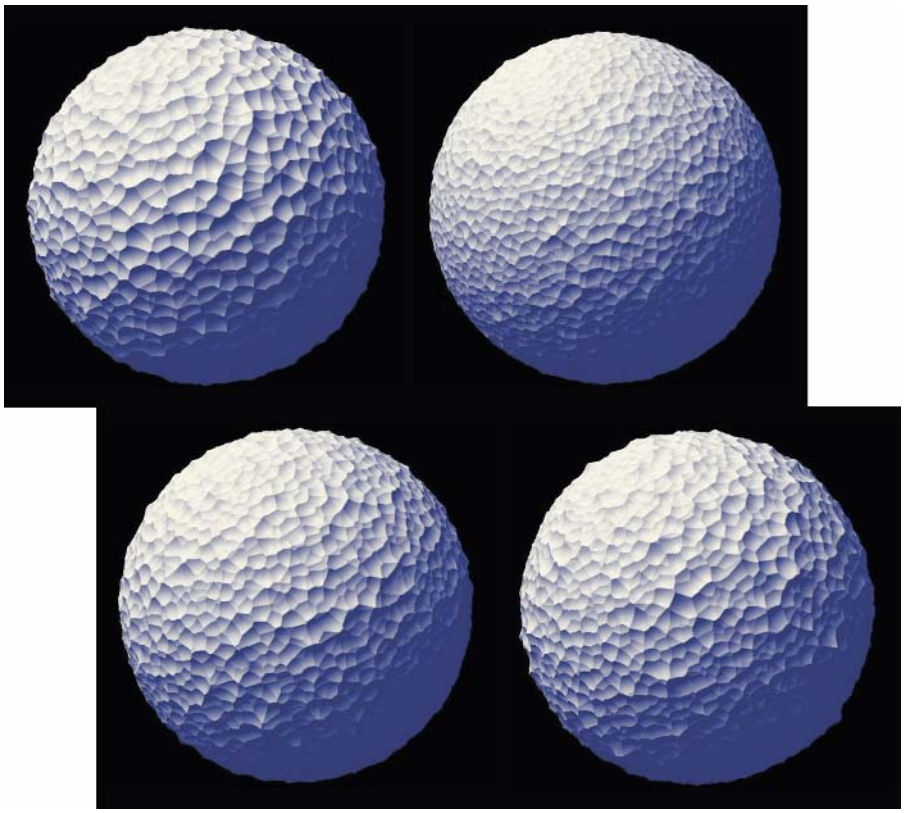


FIGURE 20.28 Planets made from Voronoi first neighbor with, going clockwise from upper left, seed tables with 1 seed per cell; 4 seeds per cell; 0–4 per cell, uniform distribution; and 0–4 per cell, Gaussian distribution.

Monofractals

Much like the various possible distributions of random numbers I just glossed over so quickly, there are even more complicated mathematical measures that characterize the randomness in random fractals. I'll do my best to simplify and clarify the complicated and obscure here, so please bear with me.

Early fractal terrains were derived from a mathematical function called *fractional Brownian motion* (fBm). fBm is a generalization of Brownian motion, which

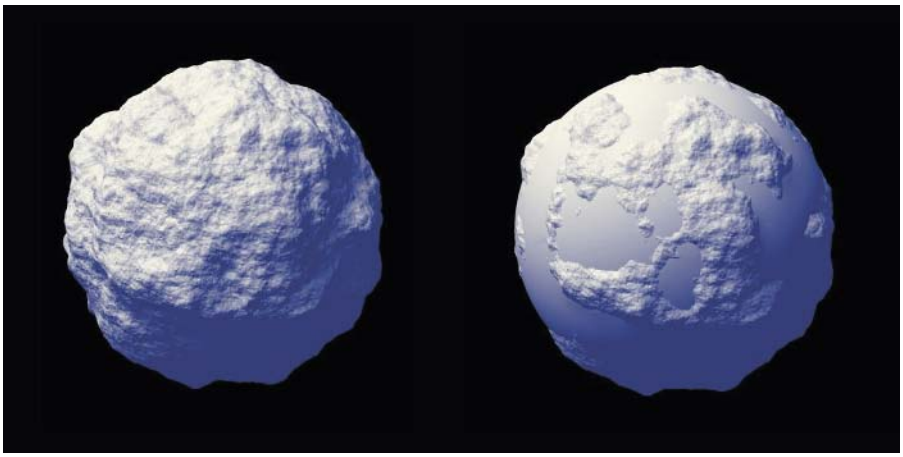


FIGURE 20.29 A planet made from monofractal fBm, a Perlin basis, and roughness of 0.3, with and without "oceans."

you may remember from your high school science classes: Brownian motion is the random walk of a very small particle being jostled about by the thermal motions of the much smaller particles comprising the fluid in which it is suspended. It's a lot like the random walk of an aimless drunk staggering about on a flat plain. fBm has a bunch of specific properties, and foremost among them is its uniformity: it's designed to look the same at all places and in all directions.[3] You can think of it as "statistically monotonous"—hence the name *mono*fractal.

This was a good start, and variations on fBm, generated from a variety of different basis functions, remain the standard random fractal used in MojoWorld and other fractal programs that create models of natural phenomena like mountains, clouds, fire, and water (see Figure 20.29).

Multifractals

Real terrains are not the same everywhere. Alpine mountains can rise out of flat plains—the eastern margin of America's Rocky Mountains being a conspicuous example. Early in my work with Mandelbrot, I wanted to capture some more of

3. In math lingo, this property is called *statistical stationarity.*

that kind of variety in fractal terrains, without complicating the very simple mathematical model that gives us fBm. As usual, I was reasoning as an artist, not a mathematician. I had some ideas about how the fractal dimension, or roughness, of terrain should vary with altitude and slope. For example, I knew that lakes fill in with sediment and become meadows as geologic time passes, so I thought, "Low areas should remain smooth, while peaks should be rough." Interestingly, the opposite appears to be more common in nature, but what the heck, I was working in a dark closet (no kidding) in the Yale math department at the time, so I was just working from memory, not active field work. I fiddled with my math—more, with my programs that implemented the math.

In order to escape Yale with my Ph.D. and not get roasted like a pig on a spit at my thesis defense, I tightened up my descriptions of these multifractal functions and did some interesting experiments that led to dead ends, from the standpoint of my ultimate goal, making MojoWorlds. I did get a little attention from some physicists interested in multifractals, which I thought was cool, as an artist. At any rate, I packaged up the two best-behaved multifractals I had devised and stuck them in MojoWorld.

"What do I care?," you ask. Well, monofractals get pretty boring pretty fast because they're the same everywhere, all the time. Multifractals are a little more interesting, visually: they're *not* the same everywhere. They're smoother in some places and rougher in others. Nature, of course, is far more complex than this, but hey, it's a second step in the right direction. In general, I use multifractals for my terrains and usually use monofractals for clouds and textures.

The Heterofractal Function

The first of the two multifractal functions in MojoWorld I call *heterofractal*. Its roughness is a function of how far it is above or below "sea level" (where it tends to be quite smooth and boring). You can add in another function to a heterofractal terrain to move the terrain around vertically, so that the smooth areas don't always occur at the same altitude (see Figure 20.30).

The Multifractal Function

The second of the two multifractal functions in MojoWorld is simply named *multifractal*. Back when I was still trying to figure out the Byzantine mathematics of multifractals, as in my dissertation and the first edition of this book, I called this

Team LRN

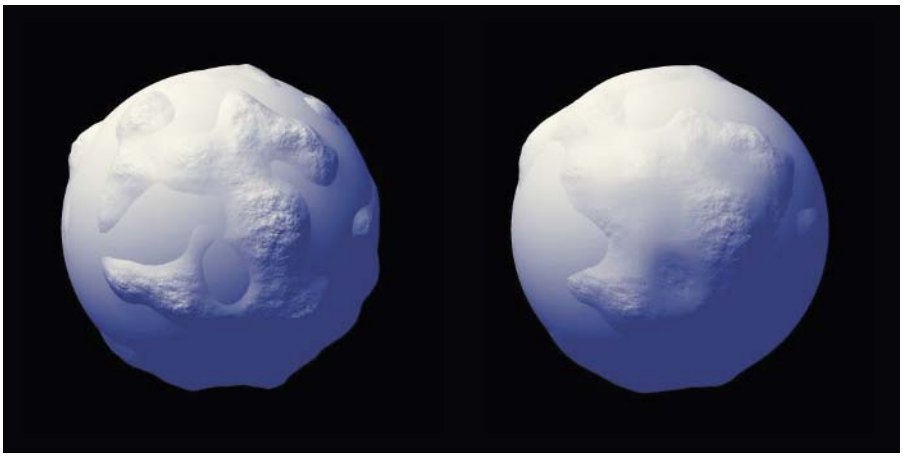


FIGURE 20.30 Planets made from a heterofractal with a Perlin basis and an "ocean" sphere at sea level. Straight heterofractal on the left, heterofractal plus a random vertical displacement on the right.

function a "hybrid multifractal," for technical reasons. What I was then calling "multifractal" turned out to be a dead end, for practical purposes, so I've since promoted "hybrid multifractal" to "multifractal" in my personal lexicon, and in MojoWorld's, by transitivity. It's a good workhorse and my first choice for terrains on a planetary scale most every time (see Figure 20.31).

Function Fractals

A fractal consists of some form repeated over a range of scales. There is no inherent restriction on what that form might be. In the good, safe practice of image synthesis, however, there are some fundamental restrictions. More math being glossed right over: There is a highest spatial frequency that can be used when synthesizing images on the computer. There is a mathematical theorem that tells us what that highest frequency is—the *Nyquist sampling theorem.* Frequencies higher than that limit, the *Nyquist limit,* will *alias* (turn into undesirable artifacts) in our images. The upshot: You don't want to go building fractals out of just anything; you really want to know what the highest spatial frequency is in your basis function. But we know you most likely care more about making interesting pictures than hearing about mathematical limitations. Hence we have *function fractals* in MojoWorld (see Figure 20.32). Rather than limiting you to using the carefully crafted basis functions built by well-

Team LRN

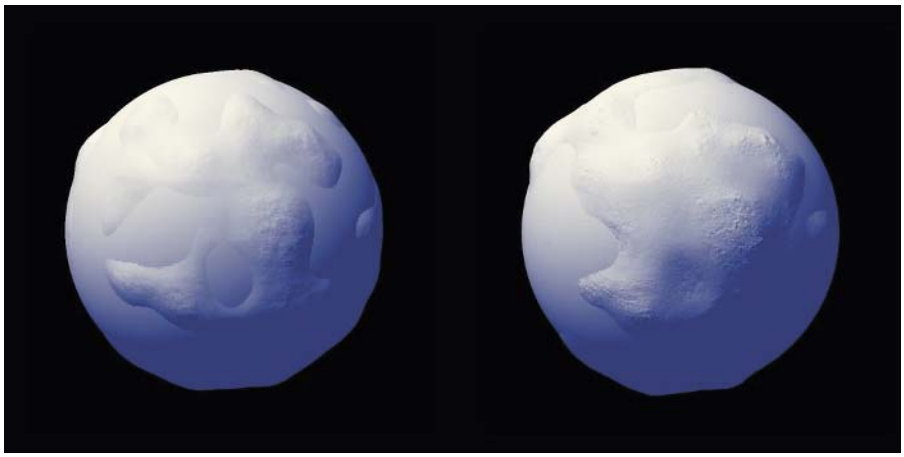


FIGURE 20.31 Planets made from multifractal 3 with a Perlin basis and a smooth sphere at sea level. Straight multifractal on the left, multifractal plus a random vertical displacement on the right.

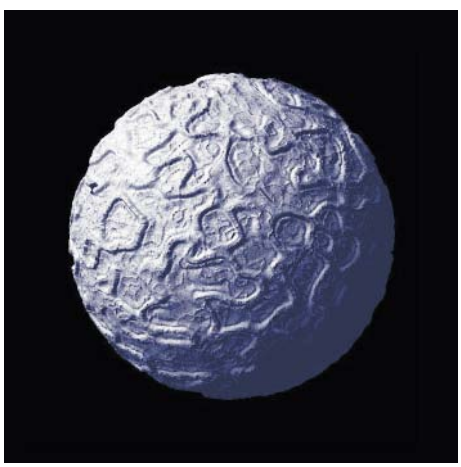


FIGURE 20.32 A planet made from a 3D function fractal.

informed professional engineers, with function fractals you can build them out of whatever ill-informed, ill-advised basis function you can come up with.

But be warned, there are two predictable consequences of using function fractals: First, they will tend to be slow. The arbitrary basis function you put in may already be a fractal that requires thousands of CPU cycles to evaluate. Repeat that at a

hundred different scales, and rendering your image will redefine the word "slow." Second, such fractals are very likely to alias badly. Such aliasing will usually look kind of like sandpaper in a still image—not too evil. But when you animate it, it can sizzle in a most annoying way. Also, the way MojoWorld is designed, if you have aliasing anywhere in your image, you're likely to have it everywhere, whereas in ordinary 3D graphics software aliasing generally increases with distance.

Don't say we didn't warn you. But then, we have a motto here at Pandromeda: "Give 'em enough rope."

Domain Distortion

When I first started playing with procedural textures like we're using in MojoWorld, one of the first things I tried is distorting one texture with another. Because we accomplish this by adding the output of one fractal function to the input of another, we call it *domain distortion*. Imagine a function with a one-dimensional domain, say, the sine wave. Undistorted, it looks like Figure 20.33.

Imagine adding the cosine to x before we pass x to the sine function. What we're computing is then not the sine of x, but the sine of x plus the cosine of x. As the value of the cosine function varies from -1.0 to 1.0 and is added to x, it has the effect of displacing the x value that gets passed to the sine function, moving it back and forth from its undistorted value (see Figure 20.34). Distorting the input of a function has the effect of distorting the output. We see such a result in Figure 20.35.

Of course, this example is what mathematicians call "trivial." It's just to illustrate the process simply and clearly. Domain distortion in MojoWorld will generally involve more complex functions and take place in higher dimensions, but the way it's done is exactly the same. Figure 20.36 shows planets made from an undistorted fractal and the same fractal with domain distortion. The domain distortion has the effect of stretching the distorted texture out in some places and pinching it together in others.

MojoWorld's Texture Editor allows domain distortion both to the basis functions and to the aggregate fractal. For efficiency and to avoid severe aliasing, basis

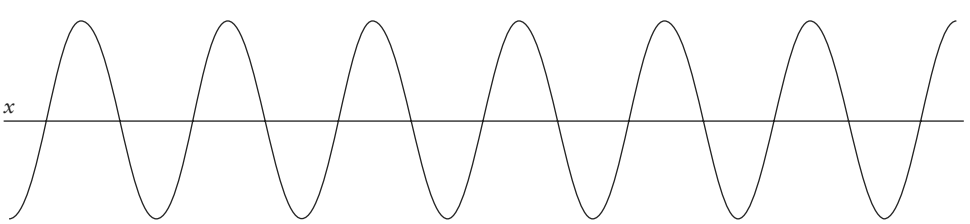


FIGURE 20.33 An undistorted sine wave: $y = \sin(x)$.

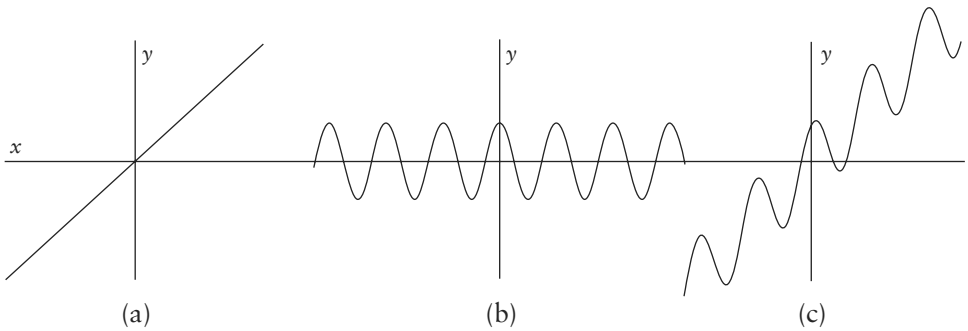


FIGURE 20.34 (a) The undistorted domain $y = x$, (b) a distortion function $y = \cos(x)$, and (c) the distorted domain $y = x + \cos(x)$.

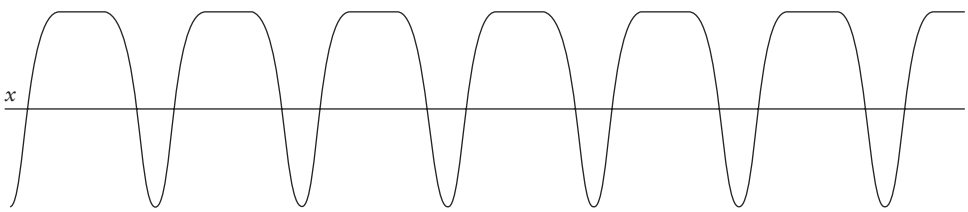


FIGURE 20.35 A distorted sine wave $y = \sin(x + \cos(x))$.

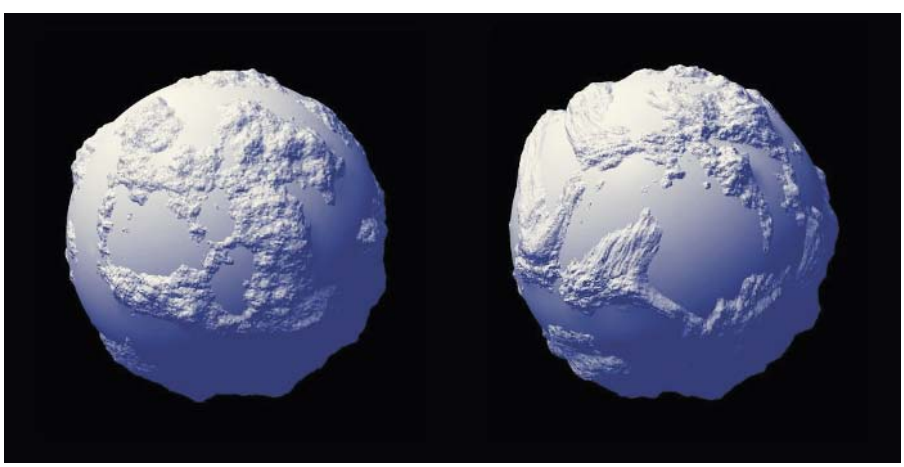


FIGURE 20.36 A planet with undistorted and distorted fractal relief.

functions can only be distorted with other basis functions. On the other hand, fractal functions can be distorted with other fractals—indeed, with any function. There are at least two reasons for the two being treated differently. The first is that, for three-dimensional functions, the distortion function is three times as expensive to evaluate as the distorted function. This is because the distortion function has to be evaluated once for each of the three input dimensions to the distorted function, while the distorted function only has to be evaluated once, using that distorted three-dimensional input. The second is that the distortion function has to be evaluated once per octave of the distorted function, in the case of distortion applied to the basis function, but only once for distorting the aggregate fractal. As MojoWorld is churning through around 25 to 30 octaves—and potentially many more—when you're down close to the surface of a planet, performing anything other than simple distortion on a per-octave basis is simply not computationally practical—yet. No doubt it will become so in years to come, with faster computers, but it will still present aliasing problems.

Distorted Fractal Functions

You can use domain distortion to make long, linear mountain ranges in the areas where the distorted texture, stretched in one direction and pinched in the other. But when you zoom in to those mountains, it's not very realistic to have all the little details stretched and pinched in the same way—real mountains just don't look like that. So I borrowed an idea from turbulent flow, *viscous damping,* to get around that. Turbulent flow is damped, or slowed, by viscosity in the fluid at small scales.[4] Since domain distortion is kind of like turbulence, I thought, "Why not do the same thing with the domain distortion? Taper it off at smaller scales." So MojoWorld has what I call *distorted fractal functions* available in the Graph Editor, or Pro UI. These are complicated little beasts—definitely a very advanced feature. Beginners will find them hopelessly confusing, I fear. But they have two fields, *onset* and *viscosity,* in which you can specify where the viscous damping begins and is total (no distortion), respectively. The scales are specified in meters, the default unit of scale in MojoWorld.

Crossover Scales

This idea of scales, for the onset of viscosity and where viscous damping is complete, leads to our next and last advanced feature in MojoWorld fractals: *crossover scales*.

4. "Small" is a relative term here; the scales at which viscosity damps turbulence depends on the viscosity of the turbulent fluid, which can be anything from molten rock, as in plumes in the Earth's mantle, to the tenuous gas in a near-perfect vacuum that we see in Figure 20.6.

Team LRN

A crossover scale is simply a scale where the behavior of a fractal changes. The simplest examples are the *upper crossover scale* and *lower crossover scale,* above and below which fractal behavior vanishes.

MojoWorld has such crossover scales. There's always a largest feature size for any MojoWorld fractal, and if you don't explicitly set a lower crossover scale, MojoWorld will eventually say "enough" and quit adding detail. (That limit is up to the MojoWorld programmers.) In the *distorted fractal* functions we employ another kind of crossover scale, in another very advanced MojoWorld feature. I figured that zooming in on a single fractal, from the scale of continents to that of dirt, is neither very interesting nor realistic. In Nature, the character of terrain is different at different scales. So, in MojoWorld's distorted fractals, I included the ability to use different basis functions at different scales. You can use up to three different basis functions in these fractals. When you use more than one, you have to specify the scale, in meters, where the crossover between basis functions begins and ends. It's not easy to show how this works, other than in an animated zoom. So please accept my apologies; no illustration here.

Driving Function Parameters with Functions

One of the most powerful features of the MojoWorld Graph Editor, or Pro UI, is the ability to drive the value of almost any parameter of any function with the output of any other function. This can give some really wild and complicated results! Once you've become an advanced MojoWorld user, I recommend going into the Pro UI and playing with this. (It will probably be hopelessly confusing until you learn to think in the new, purely procedural MojoWorld paradigm.)

For example, using a "blend" node you can easily make a texture whose color is white above a certain altitude—the snow line. But a straight, horizontal snow line is not very natural-looking. So you might create an "add" node, with "altitude" as one input and a fractal as the other. The add node is then a function with inputs and an output. You can hook the output of the add node to the parameter that controls where the blend node makes the transition to white. Now the snow line will be fractal and quite natural-looking!

A potent hidden aspect of the MojoWorld Graph Editor is that each node knows the dimensionality of the input it needs. All nodes will automatically provide output of the dimensionality requested by the parameter they are hooked into. Note that this doesn't free the user from having to make the right choice of dimensionality for certain function modes, as MojoWorld can't know what kind of effect you're out to create, and so it can't always make the choice for you.

Team LRN

USING FRACTALS

I've talked a lot about fractals and all their wonderful complexities in MojoWorld. Now let me talk a little about the specific uses for fractals.

Textures

Perhaps the main use for fractals in MojoWorld is in surface textures. Sure, MojoWorld can create some very complex geometry, but you can still get most of your interesting visual information from textures applied to surfaces. The procedural methods used in MojoWorld were originally designed by Darwyn Peachey and Ken Perlin for creating such surface textures. You can create some really beautiful effects in color alone, using fractal procedural textures. See Figures 20.37 and 20.38 for examples from MojoWorld.

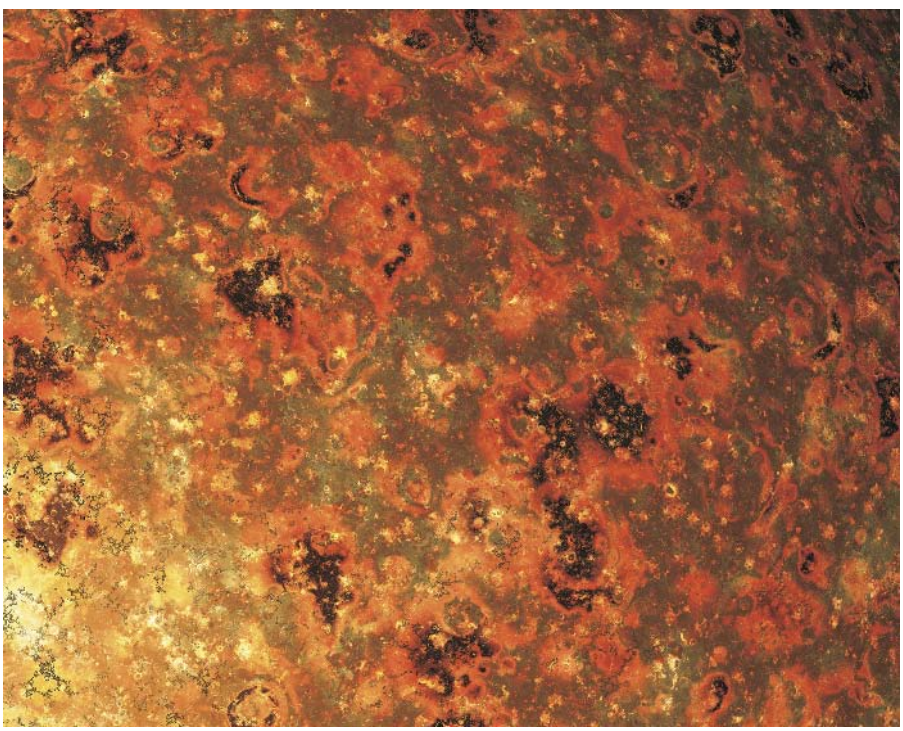


FIGURE 20.37 This MojoWorld texture study by Jonathan Allen illustrates the kind of complex and abstract beauty that can issue from procedural textures. Copyright © 2002 Jonathan Allen.

FIGURE 20.38 Another MojoWorld texture study: Complex behaviors result from fractals constructed from different basis functions, blended using various combination modes, with surface displacements for additional subtle detail. This and all the MojoWorld images in this chapter can be reproduced from the mjw files available at this book's Web site.

The development and artful use of fractal textures has pretty much made my career. Such texture functions can be used to control surface color, shininess, displacement, transparency—you name it. Pretty much everything that makes a MojoWorld interesting and beautiful is some form of a fractal procedure, or *procedural texture*. That's why the Texture Editor is the very heart of MojoWorld. Doc Mojo's advice: Spend time mastering the Texture Editor. It is by far the most powerful tool in MojoWorld.

Terrains

Even the terrains that comprise a MojoWorld's planetary landscape are just procedural textures, used in this case to determine elevation. For consistency, the terrain

texture is evaluated on the surface of a sphere of constant radius, and the result is used to raise or lower the planet's terrain surface. The multifractal functions in MojoWorld were originally designed for modeling terrain, so I recommend using them when you're creating the terrain for a MojoWorld.

Displacement Maps

MojoWorld also features *displacement maps*—textures that can actually displace the surfaces they're applied to. Yes, a MojoWorld is just a displacement-mapped sphere. But the algorithm used to displace the planet's surface is a special one, designed for speed (at the expense of memory). There are two consequences to this: First, you can make displacement-mapped spheres for moons, but you can't zoom into them like you can a MojoWorld. (Well, you *could,* but the renderer would crawl to a halt, as you got close. We'll fix this in a future version.) Second, you can do lateral displacements on the MojoWorld terrain to get overhangs. Figure 20.39 shows an example of this. This is a majorly cool feature of MojoWorld, although extreme

FIGURE 20.39 *Peeking* demonstrates an extreme example of displaced displacements in MojoWorld: first, the planet sphere is displaced using sparse convolution to create the spires, then a Voronoi-based displacement material is applied to the surface, creating overhangs on the near-vertical surfaces. Copyright © 2002 Irene Alora.

displacements can throw MojoWorld's photorealistic renderer, producing unpredictable artifacts. MojoWorld's real-time renderer can only handle one displacement—that of the terrain on the planet sphere—so such lateral displacements aren't visible there at all.

Clouds

MojoWorld 1.0 Near Space only has two-dimensional clouds, mapped onto spheres concentric with the planet. You can put any texture you like on those spheres to represent your clouds. (You can do some really wild clouds—go for it!) Future versions will have full volumetric three-dimensional clouds. "We have the technology." See the "Great Balls of Fire" section of my Web site—*www.pandromeda.com/musgrave*—for some animated examples of what I've done in the past. We'll include those and more in future versions of MojoWorld.

More exciting still, in a future version we'll put in four-dimensional clouds. This will allow you to animate volumetric clouds over time. That will be fun!

Planets

Obviously, MojoWorld is an exercise in modeling entire planets with fractals. The possibilities are endless. I'm looking forward with great anticipation to seeing what people create and find in Parametric Hyperspace. They're all out there, virtually, just waiting to be discovered. Figure 20.40 is one of may favorite early examples.

Nebulae

Figure 20.6 illustrates rather convincingly that astrophysical nebulae can be fractal in nature. Figure 20.41 illustrates an early experiment of mine in modeling with a multifractal texture—interstellar dust clouds like we see in the dark lanes in the Milky Way.

Planets reside in solar systems, solar systems in galaxies, and galaxies in clusters. In future versions of MojoWorld we plan to model all these things. Volumetric nebulae are something we're all looking forward to seeing, playing with, and zipping through!

THE EXPRESSIVE VOCABULARY OF RANDOM FRACTALS

People often ask me, "Can you do people with fractals?" The answer is no. Not everything is fractal. People don't look the same on a variety of scales (although parts of us, like our lungs and vascular systems, do). There is a limit to what can be done

FIGURE 20.40 *Planet Copperwine* is probably my favorite early MojoWorld image. The fantastic terrain, colors, moon atmosphere, clouds, and sea all make this a world that calls to me in a deep way. Copyright © 2001 Armands Auseklis.

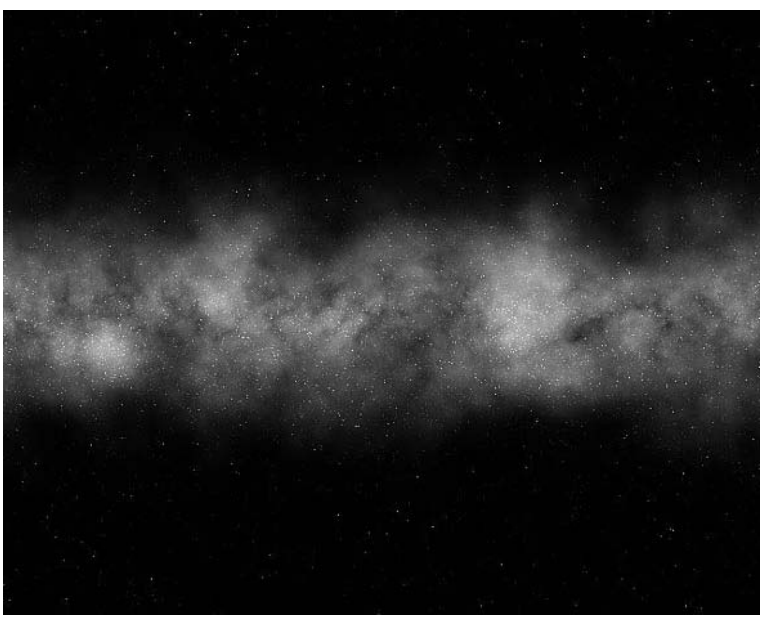


FIGURE 20.41 A multifractal texture as a model of the Milky Way.

with fractals. We can do a lot, but certainly far from *everything*. MojoWorld is designed to do most of what can be done with random fractals today. So while we can't do everything we'd ever want to do in MojoWorld, what we *can* do will keep us busy and entertained for some time to come. MojoWorld reveals a rich, new creative space for us to explore, and that exploration will keep us entertained for the rest of my life, I'm quite certain.

EXPERIMENT!

Get in there and experiment. Although Parametric Hyperspace as spanned by MojoWorld version 1.0 certainly doesn't include every world we'd ever like to see and explore, it does span an infinitely vast virtual universe, much larger in fact than the one we inhabit, simply because it has so many dimensions. Get in there and find/create cool images/places and *share them* with the rest of us! You never know:

FIGURE 20.42 *Dale—Winds of Interference* illustrates a MojoWorld—derived from *Planet Copperwine*—that calls for exploration, imaging, and perhaps colonization. Copyright © 2002 Armands Auseklis.

FIGURE 20.43 The parallel universe of Parametric Hyperspace is a place where your imagination can run wild. *Charon* is an example of the kind of place that you can construct/find there. Copyright © 2002 Armands Auseklis.

planets are big places; someone else is very likely find a more beautiful view on a planet you've created than any you've yet found. If you become a master builder of MojoWorlds, you won't have the time you need to explore them properly. You'll need help. We foresee at least two kinds of MojoWorld users: creators and explorers. The creators will do the work of constructing new planets, while the explorers will be the landscape photographers, the journalists who find and document in images the beauty in these worlds. A true team effort—that's another fun aspect of MojoWorld.

THE FUTURE

We've put into MojoWorld everything I can think of that's practical and even a few things—like the "distorted fractal" functions—that probably aren't. Yet. What's practically doable on your home computer is a fast-moving target: they get faster at an amazing rate. We've designed MojoWorld to push the state of the art. And

MojoWorld can bring any processor in the world to its knees quite easily. Of course, we have other algorithms like radiosity (superaccurate illumination calculations), physically based erosion, and fluid dynamics that we could stick in MojoWorld, just in case we need to slow things down some more.

The Holy Grail we seek is virtual reality. Not the lame, anything-but-real stuff we've seen called "VR" to date, but *believable* virtual reality—interactive MojoWorlds as beautiful and realistic as those we can render currently in non–real time. It will happen. And not too long from now. It's only a matter of engineering.

Team LRN

21

ON THE FUTURE: ENGINEERING THE APPEARANCE OF CYBERSPACE

F. KENTON MUSGRAVE

> Cyberspace. A consensual hallucination experienced daily by billions of legitimate operators, in every nation, by children being taught mathematical concepts . . . A graphic representation of data abstracted from the banks of every computer in the human system. Unthinkable complexity. Lines of light ranged in the nonspace of the mind, clusters and constellations of data. Like city lights, receding . . .
>
> —William Gibson, *Neuromancer*

INTRODUCTION

This chapter derives from an invited paper delivered to the Computer Graphics International Conference in Calgary, Canada, in 1999. It makes a nice epilog to this book, as it points out that procedural methods are key to the future of the human-computer interface. Or so the author would have us believe . . .

It was written before the MojoWorld project was begun in earnest. MojoWorld version 1.0 marks the first step toward the kind of procedural cyberspace advocated here. One of the most interesting things about MojoWorld is that it shows clearly, in one application program, the engineering gap that remains to be closed between what we can do in real time today, as illustrated by MojoWorld's real-time renderer, and what we want to do in real time in the future, as illustrated by MojoWorld's photorealistic renderer. The imagery generated by the real-time renderer is a pale approximation to that of the photorealistic renderer, which is in turn "good enough" to generate the alternate reality that is cyberspace. "Good enough" is, of course, a value judgment, and everyone is entitled to his or her own opinion about it. And I've started—but was too impatient to finish—a MojoWorld rendering that would have taken about 100 days to complete. (Granted, that was at a resolution of 15,000 by 10,000 pixels, far higher than required for a useful interactive cyberspace.) When we can get the quality of MojoWorld's photorealistic renderer in

Team LRN

real time at a resolution of 2000–4000 pixels on a side, we'll have the environment that will serve as cyberspace. Then we'll face the challenge of representing the data that we go into cyberspace to access in real time as well. Clearly, we have some years of work cut out for us to get from here to there.

It's interesting to keep in mind that what we're ultimately up to with MojoWorld is not simply generating realistic and beautiful imagery but a transparent user interface.

Gibson's definition of cyberspace in a science fiction novel in 1984 was remarkably prescient. Today's World Wide Web is the beginning of the organization of, and access to, the data Gibson refers to, but as a visualization it is embryonic and, so far, unremarkable—it is simply hypertext, not the immersive environment envisioned by Gibson. The fact that we are constrained to such tame representation is primarily attributable to the two separate problems of transferring large amounts of data rapidly and of computing high-quality imagery in real time. When we achieve near-instant access to the requisite data and the capacity to generate detailed immersive, interactive visuals on the fly, cyberspace will reach its vaunted potential as a useful, immersive virtual reality. Cyberspace is meant to be nothing less than the human-computer interface of the future.

But what should it look like once we've engineered the capacity to create it? Gibson's vision is rather limited here. Lines of light and glowing blue pyramids provide nice imagery for a science fiction novel, but what we require is a highly functional and truly powerful design for this new user interface. It is well established that the main conduit of information from the senses to the human brain is through vision. Thus we are going to need a carefully engineered and well-justified visual manifestation of cyberspace to fully tap its potential to organize and convey the information it embodies.

Navigation of the representation is a key issue. Early experience with virtual reality indicates that navigation of synthetic 3D spaces is difficult for humans. Keeping track of your orientation and relative and absolute positions in the synthetic environment are notoriously difficult, perhaps because of the alien appearance of the environment, its simplicity (read: lack of visual cues) compared to reality, and/or the limited engagement of our senses in current VR technology. Or perhaps it is because we, as biological entities, are better designed to deal with the kind of environment in which we evolved than with abstract, synthetic spaces.

CLAIMS

We have a biological heritage: primates roaming forests and the savanna. Invoking the neural net model, our "wetware" may be inherently optimized for certain image

recognition tasks. Primary among these is apprehending nonverbal communication of other individuals of our species: we feature exquisite sensitivity to the nuances of people's facial appearances and the dynamics thereof. External to our troop of primates, we are engineered by evolution to deal effectively with features of our natural environment: terrain and other natural phenomena.

So what is the best visual manifestation of cyberspace? It should be familiar and suited to the hardwired specializations of our neural net. That is, it should appear like nature as our ancestors over the eons knew it, which was far more intimately than modern humans.

What are the fundamental features of this "natural" cyberspace? Of course, there are the standard features of a landscape scene: terrain, clouds, atmospheric effects, water, and so on. Realism in these has been, and will continue to be, addressed to ever greater satisfaction in the literature of computer graphics research. Those are the obvious phenomena. I wish to point out some more subtle, yet equally essential, requisite features of a "natural" cyberspace:

1. It should be locally two-dimensional, like the surface of the earth.
2. On intermediate scales it should be spherical, like a planet.
3. On the largest scale it should be three-dimensional, like the universe we inhabit.
4. The geometry used for its representation should be, for the most part, fractal.

Let me now motivate and justify these claims.

The locally planar appearance of our planet is compelling enough that the "flat earth" model held sway as a sufficient model of Earth quite recently in our cultural history. Furthermore, we primates generally enjoy mobility only in two-and-one-half dimensional space, that is, within a slab extending about three meters above the surface of the earth (with occasional forays into trees, multistory buildings, and such). Also, which way is "up" is rarely in doubt. These factors greatly simplify our navigational challenges. Experience with the Bryce synthetic landscape generation software product indicates that naive users find Bryce much easier to learn and operate than general 3D graphics applications. We conjecture that this may be because (1) the user is generally dealing with a 2½D space rather than full 3D, and (2) the default natural environment features a horizon and gradated-by-altitude sky, which make it obvious which way is "up." Given this familiar and nonthreatening initial environment, a new user with no prior experience in 3D graphics naturally makes straightforward progress in learning the more recondite aspects of general 3D graphics without even being aware of how difficult mastering them can be.

Team LRN

Simply stated, 2½D environments appear to be natural and intuitive for the average person.

The global context for landscapes is a planet: a sphere, a globe. This alone could justify the claim that "natural" cyberspace should be spherical at intermediate scales. But there is another advantage to this geometry: it is topologically convenient. A spherical surface is finite but unbounded. This means that we can have our locally planar geometry, without worrying about falling off the edge of our necessarily finite world. It can also be a convenient boundary condition assumption for simulations such as artificial ecosystems. Finally, it can obviate, through spatial disconnection, the problem of engineering transitions between disparate models. Thus, for instance, no one would need to labor over making a seamless transition between the stylized appearance of an environment designed for interactive shoot-'em-up game play and that of an environment designed to host a set of non-real-time, scientifically accurate simulations of natural processes.

Planets, in turn, reside in the familiar context of three-dimensional space. Using three dimensions yields the maximum usable space by employing the highest usable dimension—human intuition being ill equipped to deal with higher-dimensional spaces. Three-dimensional space is obviously the way to organize our layout on the largest scale not only because it maximizes the total usable space but also because it can vastly increase the local density of information (this being only occasionally a virtue). Maintaining the argument that nature's appearance will always be the most comfortable, familiar, and efficacious of our visual options, the obvious way to lay out our universe of planets is to imitate the universe as we know it: planets reside in solar systems, solar systems in galaxies, galaxies in clusters, and clusters in superclusters. The real universe is inhomogeneous on large scales: there are vast voids between concentrations of galaxies. This, fortunately, can correspond to the good practice in visual design of judicious use of empty space (so-called negative space) to reduce clutter and guide the viewer's attention. That is, the sheets-and-voids distribution that characterizes the largest-scale structure of matter in the visible universe just might lend itself naturally to good visual design and effective imposition of (more or less arbitrary) order on the vast quantities of data that cyberspace is designed to represent for human consumption.

THE FRACTAL GEOMETRY OF CYBERSPACE

We must choose a geometry to use in constructing the visual representation of cyberspace. We have two realistic choices: Euclidean or fractal. The passage where Gibson defined "cyberspace" evokes a primarily Euclidean visualization: "lines of

Team LRN

light . . . like city lights, receding . . .” Auspiciously, he also mentions “clusters and constellations of data,” evoking the fractal geometry that generally better characterizes the forms found in nature. We’re all familiar with the shapes of Euclidean geometry: lines, planes, spheres, cubes, cones, and so on. Euclidean geometry is excellent for describing things made by humans and generally poor for describing the complex forms common in nature. The opposite is true of fractal geometry. You might conclude that, since cyberspace and everything in it is a human-made artifact, Euclidean geometry is the obvious choice for its visual representation. This is probably so, *for the artifacts representing the information content* in cyberspace. That is, there will be cities and schematics of devices and text and such on the planets that comprise cyberspace; these should be represented in the familiar Euclidean way. What is better made fractal is the visual *context* for that content. I maintain that the context should be like nature, and nature is largely fractal. Yet it’s not that simple and clear-cut, either.

I assume that you are familiar with Euclidean geometry; let me now give a brief overview of fractal geometry. Random fractals or so-called scaling noises such as fractional Brownian motion (Peitgen and Saupe 1988) characterize many structures in nature. Deterministic fractals such as the famous Mandelbrot set, or M-set, and the von Koch snowflake constitute another class of fractals. (Interestingly, certain deterministic fractals such as the von Koch snowflake and the Peano curve (Mandelbrot 1982) are even locally Euclidean, e.g., are comprised of straight line segments.) Fractal geometry can be most succinctly characterized as *dilation symmetry,* or invariance (perhaps only statistical) over changes in scale. That is, zooming in and zooming out, you see pretty much the same thing at different scales; appearance remains similar, hence the term *self-similarity* is used to characterize fractals. Fractal objects appear complex, due to the amount of detail evoked by this repetition of form over a range of scales. This apparent complexity may be deceivingly simple, however, as both the basic form and the rules of repetition may be very simple. How, then, do fractals mix with generally simpler Euclidean forms in our synthetic universe?

We primates are social beasts, and there will inevitably arise cyberManhattans: “hot properties,” local spots where “everyone will want to be” in the vast cyberuniverse. Again, to maintain the analogy with real cities, we will probably construct them using Euclidean geometry. Yet despite its complexity and the fact that it is not generally a good language for human-made form, fractal geometry has applications in constructing cybercities: as in real cities, space in cybercities will be at a premium. Just because your company grew from two employees to owning Microsoft doesn’t mean you can necessarily expand your corporate headquarters in

Team LRN

cyberSeattle to occupy half of downtown—that space will already be occupied. But scale is an entirely arbitrary concept in cyberspace; there is no standard meter, no inherent size to anything. Thus it won't matter how "large" you loom in cyberspace, but rather how much information you have that people want to access. Using a fractal representation, you can grow "inward" by adding ever more detail that can be seen by zooming farther in to your cyberconstruct. Growing inward like this will obviate the need to occupy ever more space, to expand your hegemony by conquest or cyberimperialism. The ability to grow inward may spawn a new esthetic based—in the fine artistic tradition of contradicting contemporary values—on the idea that smaller is better, that getting the most content into the smallest space is a greater accomplishment than growing to be as large as possible. The ability to grow inward, fractally, obviates the problem of available space in our cybercities.

Two other issues indicate the use of fractal geometry in the construction of cyberspace: first, the transfer and, second, the realistic rendering of highly complex scenes. The problem of *aliasing* is too recondite to address here, but let us say that one of the main problems in rendering very complex scenes arises from the difficulties in cramming more information in the pixel grid, or *raster,* that comprises the image, than that grid can accurately represent. The problem is inevitable because of the complexity of cyberspace. Aliasing shows up as highly unnatural and objectionable artifacts in still images and worse artifacts still in moving images. Fortunately, due to their constructive or "procedural" nature, fractals can be adaptively band-limited for alias-free rendering. That is, their construction can limit their detail to that which the raster can accurately represent. Furthermore, their procedural character ensures that everyone can explore the same synthetic universe of unlimited visual richness, without the need for hypernet bandwidth or huge, mirrored database servers simply to dish up the visual representation—as opposed to the information content—of that universe. This is because the visual richness of fractal models is an emergent property of their computation. As we have seen, a relatively simple model and a small, in terms of data transfer, number of parameter values are sufficient to generate an entire world, with potentially unlimited detail. (MojoWorld files tend to be less than 100 kilobytes in size.) A beauty of the fractal representation is that the model is tiny in comparison to the visual result. The model consists only of the basic shape and the rules for its repetition; all details emerge from the computation. This solves the bandwidth problem inherent in transferring from remote repositories to users complex representations of cyberspace—the *place,* as opposed to the *content* or information accessed there—leaving the network bandwidth free for transferring that content, which is what we'll go into cyberspace for in the first place.

Team LRN

CONCLUSION

In conclusion, I'd like to reiterate that the natural, fractal universe I am advocating is not an end in itself, but rather a *context* for the information content that cyberspace is designed to make accessible to humans. Cyberspace is strictly for human consumption. The machines do not need it; they do very well with their streams of binary code. Cyberspace is a visualization tool designed to make the exponentially growing stores of information entrusted to the computers, and their inherent value, available to humans who generally do poorly at comprehending binary-encoded information.

The "natural" cyberspace I am proposing is an efficacious setting for entertainment, like games and Myst-style puzzles, for synthetic ecosystems, simulated cities and civilizations, artistic creations, meditative spaces, and raw data retrieval. The suggested hierarchy going from 3D to 2½D to 2D preserves advantages of the evolved human faculties for interaction with our environment. Extensive use of fractals imparts visual richness, compact representation, and a natural visual character also suited to our naturally evolved faculties. It is familiar and expandable. It is as politically and aesthetically neutral and noncontroversial as such an important and soon-to-be ubiquitous aspect of human life can be made. Most importantly, it has the potential to be made as beautiful as the universe we inhabit, as Figure 21.1 illustrates.

Team LRN

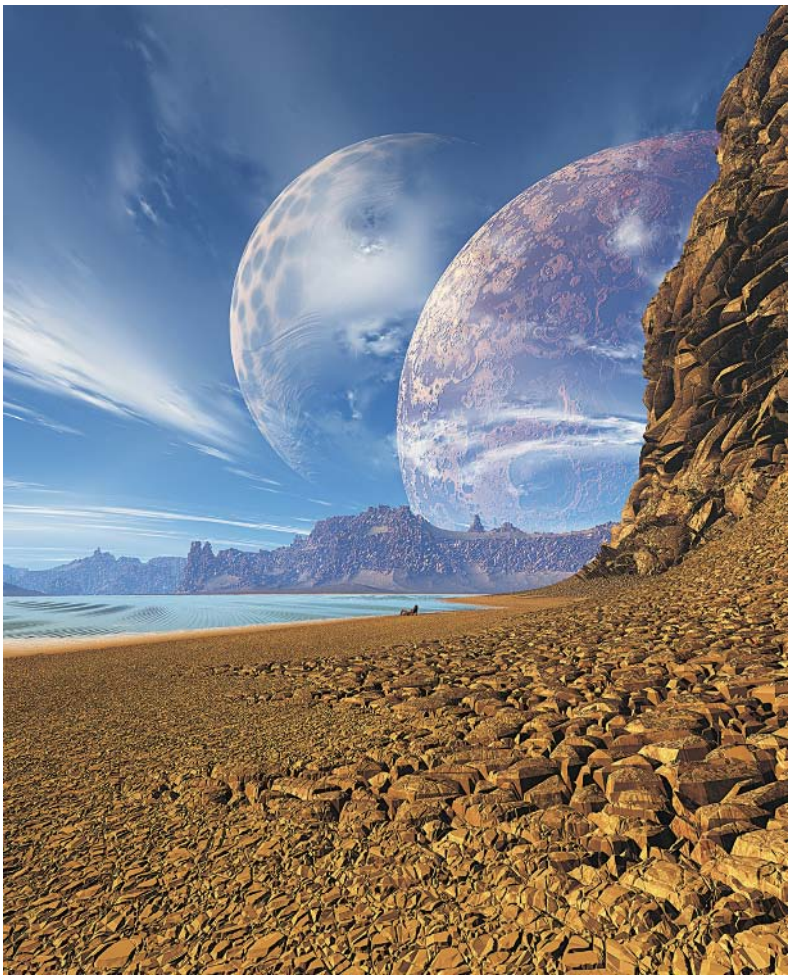


FIGURE 21.1 *Dale Beach* is a visualization of what cyberspace—or at least one tiny part of it—may look like, once we have the capability of rendering it in real time. It will, of course, have representations of the data and avatars we go there to interact with, making it in essence the ultimate human-computer interface. Copyright © F. Kenton Musgrave; world by Armands Auseklis

Team LRN

APPENDICES

APPENDIX A: C CODE IMPLEMENTING QAEB TRACING

```
    /* determines 3D position along ray at distance t */
#define RAY_POS(ray,t,pos) \
            {(pos)->x = (ray)->origin.x + t*(ray)->dir.x; \
             (pos)->y = (ray)->origin.y + t*(ray)->dir.y; \
             (pos)->z = (ray)->origin.z + t*(ray)->dir.z; }

typedef struct { /* catch-all type for 3D vectors and positions */
    double x;
    double y;
    double z; } Vector;
    /* returns TRUE if HF intersected, FALSE otherwise */
Boolean
Intersect_Terrain(int row, int column, double epsilon, Ray *ray, Hit *hit )
{
double d, /* ray parameter, equal to distance travelled */
       alt,                              /* alt at current step */
       prev_d,                           /* d at last step */
       prev_alt;                         /* alt at last step */
Vector position,                         /* current position along ray */
       prev_position;                    /* previous position along ray */

if ( row == 0 ) { /* if at bottom of bottom-to-top rendering */
    d = near_clip_dist;                  /* init march stride */
    RAY_POS( ray, d, &prev_position );   /* init previous 3D position */
    prev_alt = Displacement( prev_position, d ); /* evaluate the HF function */
} else { /* (this scheme is valid only for vertical columns) */
    d = prev_d = prev_dist[column]; /* start at final d of prev. ray in column */
    prev_position = prev_pos[column];
    prev_alt = prev_alts[column];
}

while ( d < far_clip_dist ) {            /* the QAEB raymarch loop */
    d += d * epsilon;                    /* update the marching stride */
    RAY_POS( ray, d, Sposition );        /* get current 3D position */
    alt = Displacement( position, d );   /* evaluate the HF function */
    if ( position.z < alt ) {            /* surface penetrated */
```


Team LRN

```
        Intersect_Surface( prev_alt, alt, d*epsilon, d,
        position, prev_position, ray, hit );
        prev_dist[column] = prev_d; /* update prev. distance data */
        prev_alts[column] = prev_alt;
        prev_pos[column] = prev_position;
        return( TRUE );
    }
    prev_d = d;
    prev_alt = alt;
    prev_position = position;
}

/* exceeded far clip distance; update "prev_dist" appropriately & exit. */
prev_dist[column] = d;
return( FALSE );

} /* Intersect_Terrain() */
```

APPENDIX B: C CODE FOR INTERSECTION AND SURFACE NORMAL

```
#define VEC_SUB(a,b,c)  {(c)->x = (a).x-(b).x; \
                         (c)->y = (a).y-(b).y; \
                         (c)->z = (a).z-(b).z;}
#define CROSS(a,b,c)    {(c)->x = (a.y * b.z) - (a.z * b.y); \
                         (c)->y = (a.z * b.x) - (a.x * b.z); \
                         (c)->z = (a.x * b.y) - (a.y * b.x);}
      /* assigns ray-surface intersection and surface normal */
void
Intersect_Surface( double z_near, double z_far, double epsilon, double distance,
         Vector position, Vector prev_position, Ray *ray, HitData *hit )
{
   Vector P_r,           /* point to the right, relative to ray dir and "up" */
          v_d,           /* vector from prev_position to position */
          v_l,           /* vector from prev_position to p_r */
          n;             /* surface normal */

      /* first construct three points that lie on the surface */
   prev_position.z = z_near;
   position.z = z_far;
      /* construct a point one error width to right */
   p_r.x = prev_position.x + epsilon*camera.right_dir.x;
   p_r.y = prev_position.y + epsilon*camera.right_dir.y;
   p_r.z = prev_position.z + epsilon*camera.right_dir.z;
   p_r.z = Displacement( p_r, distance );

      /* get two vectors in the surface plane; cross for surface normal */
   VEC_SUB( position, prev_position, &v_d );
   Normalize( &v_d );
   VEC_SUB( p_r, prev_position, &v_l );
   Normalize( &v_l );
   CROSS( v_l, v_d, &n );
   Normalize( &n );

      /* assign the various hit data */
   hit->distance = distance;
   hit->intersect = prev_position;
   hit->normal = n;

} /* Intersect_Surface() */
```

BIBLIOGRAPHY

Abelson, H., and A. A. diSessa. 1982. *Turtle geometry.* Cambridge, MA: MIT Press.

Abhyankar, S., and C. Bajaj. 1987a. Automatic parametrization of rational curves and surfaces I: Conies and conicoids. *Computer News,* 1:19.

Abhyankar, S., and C. Bajaj. 1987b. Automatic parametrization of rational curves and surfaces II: Conies and conicoids. *Computer News,* 3:25.

Ahlberg, J., E. Nilson, and J. Walsh. 1967. *The theory of splines and their applications.* Boston: Academic Press.

Amanatides, J. 1984. Ray tracing with cones. In H. Christiansen, ed., *Computer Graphics (SIGGRAPH '84 Proceedings),* 18:129–135.

Amburn, P., E. Grant, and T. Whitted. 1986. Managing geometric complexity with enhanced procedural models. *Computer Graphics* 20(4):189–195.

American National Standards Institute. 1986. *Nomenclature and definitions for illumination engineering, ANSI/IBS,* RP-16–1986.

Anderson, D. 1992. Hidden line elimination in projected grid surfaces. *ACM Transactions on Graphics,* 1(4): 274–288.

Apodaca, A. A. 1999. *Advanced RenderMan: Creating CGI for motion pictures.* San Francisco: Morgan Kaufmann. *See also* RenderMan tricks everyone should know in *SIGGRAPH '98,* or *SIGGRAPH '99 Advanced RenderMan Course Notes.*

Apodaca, A. A., and L. Gritz. 2000. *Advanced RenderMan: Creating CGl for motion pictures.* San Francisco: Morgan Kaufmann.

Arvo, J. 1992. Fast random rotation matrices. In David Kirk, ed., *Graphics Gems III,* Academic Press, 117.

Team LRN

Arvo, J., and D. Kirk. 1989. A survey of ray tracing acceleration techniques. In Andrew Glassner, ed., *An introduction to ray tracing,* Boston: Academic Press, 201–262.

Badler, N. I., J. O'Rourke, and B. Kaufman. 1980. Special problems in human movement simulation. *SIGGRAPH '80 Proceedings,* 189–197.

Bagby, D. 1984. Parameterization of elliptical elements. Letter to ANSI X3H3 Committee (August 16, 1984), 17.

Banks, D. 1994. Illumination in diverse codimensions. *Computer Graphics, Annual Conference Series (Proceedings of S1GGRAPH '94),* 327–334.

Baraff, D., and A. P. Witkin. 1998. Large steps in cloth simulation. *Proceedings of SIGGRAPH '98,* 43–54.

Barr, A. H. 1986. Ray tracing deformed surfaces. In Maureen C. Stone, ed., *Computer Graphics (SIGGRAPH '86 Proceedings),* 20:287–296.

Barr, A. H. 1991. Teleological modeling. In N. I. Badler, B. A. Barsky, and D. Zeltzer, eds., *Making them move: Mechanics, control, and animation of articulated figures,* San Francisco: Morgan Kaufmann, 315–321.

Battke, H., D. Stalling, and H.-C. Hege. 1996. *Fast line integral convolution for arbitrary surfaces in 3D.* Preprint SC-96–59, Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB).

Bennis, C., J. Vezien, and G. Iglesias. 1991. Piecewise surface flattening for non-distorted texture mapping. *Proceedings of SIGGRAPH '91,* 237–246.

Bian, B. 1990. *Accurate simulation of scene luminances.* Worcester, MA: Worcester Polytechnic Institute.

Birkhoff, G., and S. MacLane. 1965. *A survey of modern algebra,* 3rd edition. New York: MacMillan. Exercise 15, Section IX-3, 240; also corollary, Section IX-14, 277–278.

Bishop, G., and D. M. Weimer. 1986. Fast phong shading. *Computer Graphics,* 20(4):103–106.

Blinn, J. F. 1977. Models of light reflection for computer synthesized pictures. In James George, ed., *Computer Graphics (SIGGRAPH '77 Proceedings),* 11:192–198.

Blinn, J. F. 1978. Simulation of wrinkled surfaces. In James George, ed., *Computer Graphics (SIGGRAPH '78 Proceedings),* 12:286–292.

Team LRN

Blinn, J. F. 1982a. Light reflection functions for simulation of clouds and dusty surfaces. In *Computer Graphics (SIGGRAPH '82 Proceedings)*, 16(3):21–29.

Blinn, J. F. 1982b. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256.

Blinn, J. F., and M. E. Newell. 1976. Texture and reflection in computer generated images. *Communications of the ACM*, 19:542–546.

Bloomenthal, J. 1985. Modeling the mighty maple. In B. A. Barsky, ed., *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):305–311.

Bloomenthal, J., C. Bajaj, J. Blinn, M. P. Cani-Gascuel, A. Rockwood, B. Wyvill, and G. Wyvill. 1997. *Introduction to implicit surfaces*. San Francisco: Morgan Kaufmann.

Bouville, C. 1985. Bounding ellipsoids for ray fractal intersection. In B. A. Barsky, ed., *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):45–52.

Bracewell, R. N. 1986. *The Fourier transform and its applications*. New York: McGraw-Hill.

Breen, D. E., D. H. House, and M. J. Wozny. 1994. Predicting the drape of woven cloth using interacting particles. *Proceedings of SIGGRAPH '94*, 365–372.

Brigham, E. O. 1988. *The fast Fourier transform and its applications*. Englewood Cliffs, NJ: Prentice Hall.

Brinsmead, D. 1993. Convert solid texture. Software component of Alias|Wavefront Power Animator 5.

Brooks, R. 1986. A robust layered control for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23.

Brown, L. M., and J. Rigden, eds. 1993. *Most of the good stuff*. New York: American Institute of Physics.

Burt, P. J., and E. H. Adelson. 1983. A multiresolution spine with application to image mosaics. *ACM Transactions on Graphics*, 2(4)217–236.

Cabral, B., N. Cam, and J. Foran. 1994. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *ACM/IEEE Symposium on Volume Visualization*, 91–98.

Team LRN

Cabral, B., and L. Leedom. 1993. Imaging vector fields using line integral convolution. *Computer Graphics Proceedings, Annual Conference Series,* 263–270.

Cabral, B., N. Max, and R. Springmeyer. 1987. Bidirectional reflection functions from surface bump maps. In M. C. Stone, ed., *Computer Graphics (SIGGRAPH '87 Proceedings),* 21:273–281.

Cabral, B., M. Olano, and P. Nemec. 1999. Reflection space image based rendering. *Proceedings of SIGGRAPH '99,* 165–170.

Calvert, T. W., J. Chapman, and A. Patla. 1980. The integration of subjective and objective data in the animation of human movement. In *Computer Graphics (SIGGRAPH '80 Proceedings),* 14:198–203.

Canright, D. 1994. Estimating the spatial extent of attractors of iterated function systems. *Computers and Graphics* 18(2):231–238.

Carpenter, L. 1984. The A-buffer, an antialiased hidden surface method. In H. Christiansen, ed., *Computer Graphics (SIGGRAPH '84 Proceedings),* 18:103–108.

Carr, N. A., and J. C. Hart. 2002. Meshed atlases for real-time procedural solid texturing. *ACM Transactions on Graphics,* 21(2), 106–131.

Catmull, E. E. 1974. *A subdivision algorithm for computer display of curved surfaces*. Ph.D. thesis, Department of Computer Science, University of Utah.

Chadwick, J., D. Haumann, and R. Parent. 1989. Layered construction for deformable animated characters. *Computer Graphics,* 23(3):243–252.

Chen, L., G. T. Herman, R. A. Reynolds, and J. K. Udupa. 1985. Surface shading in the cuberille environment. *IEEE Computer Graphics and Applications,* 5(12):33–43.

Cignoni, B., C. Montani, C. Rocchini, and B. Scopigno. A general method for preserving attribute values on simplified meshes. *Proceedings of Visualization '98,* 59–66.

Cline, H. E., W. E. Lorensen, and S. Eudke. 1988. Two algorithms for the three dimensional reconstruction of tomograms. *Medical Physics,* 15(3):320–327.

Cohen, J., M. Olano, and D. Manocha. 1998. Appearance-preserving simplification. *Proceedings of SIGGRAPH '98,* 115–122.

Cook, R. L. 1984. Shade trees. In H. Christiansen, ed., *Computer Graphics (SIGGRAPH '84 Proceedings),* 18(3):223–231.

Team LRN

Cook, R. L. 1986. Stochastic sampling in computer graphics. *ACM Transactions on Graphics,* 5(1):51–72.

Cook, R. L., E. Carpenter, and E. Catmull. 1987. The Reyes image rendering architecture. In M. C. Stone, ed., *Computer Graphics (SIGGRAPH '87 Proceedings),* 95–102.

Cook, R. L., T. Porter, and E. Carpenter. 1984. Distributed ray tracing. In H. Christiansen, ed., *Computer Graphics (SIGGRAPH '84 Proceedings),* 18(3):137–145.

Cook, R. L., and K. E. Torrance. 1981. A reflectance model for computer graphics. In *Computer Graphics (SIGGRAPH '81 Proceedings),* 15:307–316.

Coquillart, S., and M. Gagnet. 1984. Shaded display of digital maps. *IEEE Computer Graphics and Applications,* 35–42.

Cotton, W., and A. Anthes. 1989. *Storm and cloud dynamics*. San Diego: Academic Press.

Crow, F. C. 1984. Summed-area tables for texture mapping. In H. Christiansen, ed., *Computer Graphics (SIGGRAPH '84 Proceedings),* 18:207–212.

Cychosz, J. M. 1986. Vectorized ray tracing of polygonal models. Unpublished results.

Dawkins, R. 1987. *The blind watchmaker.* New York: W. W. Norton.

DeGraf, B., and M. Wahrman. 1988. Mike the talking head. *Computer Graphics World,* 11(7):57.

Deussen, O., P. M. Hanrahan, B. Lintermann, R. Mech, M. Pharr, and P. Prusinkiewicz. 1998. Realistic modeling and rendering of plant ecosystems. *Proc. of SIGGRAPH '98,* 275–286.

Dippé, M. A. Z., and E. H. Wold. 1985. Antialiasing through stochastic sampling. In B. A. Barsky, ed., *Computer Graphics (SIGGRAPH '85 Proceedings),* 19:69–78.

Dobashi Y., K. Kaneda, H. Yamashita, T. Okita, and T. Nishita. 2000. *A simple, efficient method for realistic animation of clouds.* In K. Akeley, ed., *Proceedings of SIGGRAPH 2000, Computer Graphics Proceedings, Annual Conference Series,* 19–28.

DoCarmo, M. 1976. *Differential geometry of curves and surfaces.* Englewood Cliffs, NJ: Prentice Hall.

Drebin, R., L. Carpenter, and P. Hanrahan. 1988. Volume rendering. In John Dell, ed., *Computer Graphics (SIGGRAPH '88 Proceedings),* 22(4):65–74.

Team LRN

Dubuc, S., and R. Hamzaoui. 1994. On the diameter of the attractor of an IFS. C. R. Math Rep. Sci., Canada XVI 2, 3, 85–90.

Duff, T. 1985. Compositing 3-D rendered images. In B. A. Barsky, ed., *Computer Graphics (SIGGRAPH '85 Proceedings),* 19:41–44.

Dungan, W., Jr. 1979. A terrain and cloud computer image generation model. In *Computer Graphics (SIGGRAPH '79 Proceedings),* 13:143–150.

Ebert, D., and E. Bedwell. 1998. Implicit modeling with procedural techniques. *Proceedings Implicit Surfaces '98,* Seattle, WA.

Ebert, D., K. Boyer, and D. Roble. 1989. Once a pawn a foggy knight . . . [videotape]. *SIGGRAPH Video Review,* 54. New York: ACM SIGGRAPH, Segment 3.

Ebert, D., W. Carlson, and R. Parent. 1994. Solid spaces and inverse particle systems for controlling the animation of gases and fluids. *The Visual Computer,* 10(4):179–190.

Ebert, D., J. Ebert, and K. Boyer. 1990. Getting into art [videotape]. Department of Computer and Information Science, Ohio State University.

Ebert, D., J. Kukla, T. Bedwell, and S. Wrights. 1997. A cloud is born. *ACM SIGGRAPH Video Review.* SIGGRAPH '97 Electronic Theatre Program. New York: ACM SIGGRAPH.

Ebert, D., F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. 1994. *Texturing and modeling: A procedural approach.* Cambridge, MA: Academic Press.

Ebert, D. S. 1991. *Solid spaces: A unified approach to describing object attributes*. Ph.D. thesis, Ohio State University.

Ebert, D. S. 1997. Volumetric modeling with implicit functions: A cloud is born. In David Ebert, ed., *SIGGRAPH '97 Visual Proceedings,* ACM SIGGRAPH, 147.

Ebert, D. S., and R. E. Parent. 1990. Rendering and animation of gaseous phenomena by combining fast volume and scanline A-buffer techniques. In Forest Baskett, ed., *Computer Graphics (SIGGRAPH '90 Proceedings),* 24:357–366.

Ekman, P., and W. Friesen. 1978. *Manual for the facial action coding system*. Palo Alto, CA: Consulting Psychologists Press.

Elinas, P., and W. Stürzlinger. 2000. Real-time rendering of 3D clouds. *Journal of Graphics Tools,* 5(4):33–45.

Team LRN

Etzmuss, O., B. Eberhardt, and M. Hauth. 2000. Implicit-explicit schemes for fast animation with particle systems. *Computer Animation and Simulation 2000,* 138–151.

Evertsz, C. J. G., and B. B. Mandelbrot. 1992. Multifractal measures. In H. O. Peitgen, H. Jürgens, and D. Saupe, eds., *Chaos and fractals,* New York: Springer-Verlag, Appendix B, 921–953.

Faigin, G. 1990. *The artist's complete guide to facial expression.* New York: Watson-Guptill.

Fedkiw, R., J. Stam, and H. Wann Jensen. 2001. Visual simulation of smoke. In E. Fiume, ed., *Proceedings of ACM SIGGRAPH 2001, Computer Graphics Proceedings, Annual Conference Series,* 15–22.

Feibush, E. A., M. Levoy, and R. E. Cook. 1980. Synthetic texturing using digital filters. In *Computer Graphics (SIGGRAPH '80 Proceedings),* 14:294–301.

Fidducia, C. M., and R. M. Mattheyses. 1982. A linear time heuristic for improving network partitions. *Proceedings of IEEE Design Automation Conference,* 175–181.

Foley, J. D., A. van Dam, S. K. Feiner, and J. F. Hughes. 1990. *Computer graphics: Principles and practice.* Systems Programming Series. Reading, MA: Addison-Wesley.

Foster, N., and D. Metaxas. 1982. Modeling the motion of a hot, turbulent gas. In Turner Whitted, ed., *SIGGRAPH '97 Conference Proceedings, Annual Conference Series,* New York: Addison-Wesley, 181–187.

Fournier, A. 1992. Normal distribution functions and multiple surfaces. *Graphics Interface '92 Workshop on Local Illumination,* 45–52.

Fournier, A., D. Fussell, and L. Carpenter. 1982. Computer rendering of stochastic models. *Communications of the ACM,* 25(6):371–384.

Fowler, D. R., P. Prusinkiewicz, and J. Battjes. 1992. A collision-based model of spiral phyllotaxis. *Computer Graphics* 26(2):361–368.

Fu, K. S., and S. Y. Lu. 1978. Computer generation of texture using a syntatic approach. In *Computer Graphics (SIGGRAPH '78 Proceedings),* 12:147–152.

Fuchs, H., Z. M. Kedem, and S. P. Uselton. 1977. Optimal surface reconstruction from planar contours. *Communications of the ACM,* 20(10):693–702.

Fusco, M. J., and S. Tice. 1993. Character motion systems course notes. *SIGGRAPH '93 Course Notes 01,* ACM SIGGRAPH, 9–22.

Team LRN

Gagalowicz, A., and S. D. Ma. 1985. Sequential synthesis of natural textures. *Computer Graphics, Vision and Image Processing,* 30:289–315.

Gallagher, R. S., and J. C. Nagtegaal. 1989. An efficient 3D visualization technique for finite element models and other course volumes. In J. Lane, ed., *Computer Graphics (SIGGRAPH '89 Proceedings),* 23:185–194.

Garber, D. D. 1981. *Computational models for texture analysis and texture synthesis.* Ph.D. thesis, University of Southern California.

Gardner, G. 1984. Simulation of natural scenes using textured quadric surfaces. In H. Christiansen, ed., *Computer Graphics (SIGGRAPH '84 Proceedings),* 18:11–20.

Gardner, G. 1985. Visual simulation of clouds. In B. A. Barsky, ed., *Computer Graphics (SIGGRAPH '85 Proceedings),* 19(3):297–304.

Gardner, G. 1990. Forest fire simulation. In F. Baskett, ed., *Computer Graphics (SIGGRAPH '90 Proceedings),* 24:430.

Garland, M., A. Willmott, and P. S. Heckbert. 2001. Hierarchical face clustering on polygonal surfaces. *Proceedings of Interactive 3D Graphics,* 49–58.

Gedzelman, S. D. 1991. Atmospheric optics in art. *Applied Optics,* 30(24):3514–3522.

Ghezzi, C., and M. Jazayeri. 1982. *Programming language concepts.* New York: Wiley.

Girard, M., and A. A. Maciejewski. 1985. Computational modeling for the computer animation of legged figures. In B. A. Barsky, ed., *Computer Graphics (SIGGRAPH '85 Proceedings),* 19:263–270.

Goehring, D., and O. Gerlitz. 1997. Advanced procedural texturing using MMX technology. Intel MMX technology application note. *http://developer.intel.com/software/idap/ resources /technical_collateral/mmx/proctex2.htm.*

Gonzalez, R. C., and R. E. Woods. 1992. *Digital image processing.* Reading, MA: Addison-Wesley.

Gordon, D., and R. A. Reynolds. 1985. Image space shading and 3-dimensional objects. *Computer Vision, Graphics, and Image Processing,* 29:361–376.

Greene, N. 1986. Applications of world projections. In M. Green, ed., *Proceedings of Graphics Interface '86,* 108–114.

Team LRN

Greene, N., and M. Kass. 1993. Hierarchical z-buffer visibility. *Proceedings of SIGGRAPH '93,* 231–240.

Grossmann, A., and J. Morlet. 1984. Decomposition of hardy functions into square integrable wavelets of constant shape. *SI AM Journal of Mathematics,* 15:723–736.

Haines, E. 1989. Essential ray tracing algorithms. In A. Glassner, ed., *An introduction to ray tracing,* Boston: Academic Press.

Haines, E., and S. Worley. 1993. Point-in-polygon testing. *Ray Tracing News,* 2:1. E-mail available under anonymous ftp from *weedeater.math.yale.edu*.

Hall, R. A. 1989. *Illumination and color in computer generated imagery.* New York: Springer-Verlag.

Hanrahan, P. 1990. Volume rendering. *SIGGRAPH '90: Course Notes on Volume Visualization.*

Hanrahan, P. 1999. Procedural shading (keynote). *Eurographics/SIGGRAPH Workshop on Graphics Hardware. http://graphics.stanford.edu/hanrahan/talks/rts1/slides.*

Hanrahan, P., and P. E. Haeberli. 1990. Direct WYSIWYG painting and texturing on 3D shapes. *Computer Graphics,* 24(4):215–223.

Hanrahan, P., and J. Lawson. 1990. A language for shading and lighting calculations. In F. Baskett, ed., *Computer Graphics (SIGGRAPH '90 Proceedings),* 24:289–298.

Harris, M. J. 2002. Real-time cloud rendering for games. *Games Developers Conference 2002 Proceedings.*

Harris, M. J., and A. Lastra. 2001. Real-time cloud rendering. *Computer Graphics Forum (Eurographics 2001 Proceedings),* 20(3):76–84.

Hart, J. C. 1992. The object instancing paradigm for linear fractal modeling. *Proceedings of Graphics Interface.* San Francisco: Morgan Kaufmann, 224–231.

Hart, J. C. 2001. Perlin noise pixel shaders. *Proceedings of the Graphics Hardware Workshop, Eurographics/SIGGRAPH,* 87–94.

Hart, J. C., N. Carr, M. Kameya, S. A. Tibbits, and T. J. Colemen. 1999. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. *1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware,* 45–53.

Team LRN

Hart, J. C., and T. A. DeFanti. 1991. Efficient antialiased rendering of 3-D linear fractals. *Computer Graphics* 25(3).

Haruyama, S., and B. A. Barsky. 1984. Using stochastic modeling for texture generation. *IEEE Computer Graphics and Applications,* 4(3):7–19.

He, X. D., K. E. Torrance, F. X. Sillion, and D. P. Greenberg. 1991. A comprehensive physical model for light reflection. In T. W. Sederberg, ed., *Computer Graphics (SIGGRAPH '91 Proceedings),* 25:175–186.

Hearn, D., and M. P. Baker. 1986. *Computer graphics.* Englewood Cliffs, NJ: Prentice Hall.

Heckbert, P. S. 1986a. Filtering by repeated integration. In D. C. Evans and R. J. Athay, eds., *Computer Graphics (SIGGRAPH '86 Proceedings),* 20:315–321.

Heckbert, P. S. 1986b. Survey of texture mapping. *IEEE Computer Graphics and Applications,* 6(11):56–67.

Heidrich, W., and H.-P. Seidel. 1999. Realistic, hardware-accelerated shading and lighting. *Computer Graphics, Annual Conference Series (Proceedings of SIGGRAPH '99),* 171–178.

Hoehne, K. H., M. Romans, A. Pommert, and U. Tiede. 1990. Voxel based volume visualization techniques. *SIGGRAPH '90: Course Notes on Volume Visualization.*

Houze, R. 1993. *Cloud dynamics.* San Diego: Academic Press.

Inakage. M. Modeling laminar flames. 1991. *SIGGRAPH '91: Course Notes 27.*

Jaquays, P., and B. Hook. 1999. *Quake 3: Arena shader manual, revision 10.*

Kajiya, J. T. 1983a. New techniques for ray tracing procedurally defined objects. *ACM Transactions on Graphics,* 2(3):161–181.

Kajiya, J. T. 1983b. New techniques for ray tracing procedurally defined objects. In *Computer Graphics (SIGGRAPH '83 Proceedings),* 17(3):91–99.

Kajiya, J. T. 1985. Anisotropic reflection models. In B. A. Barsky, ed., *Computer Graphics (SIGGRAPH '85 Proceedings),* 19:15–21.

Kajiya, J. T. 1986. The rendering equation. In David C. Evans and Russell J. Athay, eds., *Computer Graphics (SIGGRAPH '86 Proceedings),* 20:143–150.

Team LRN

Kajiya, J. T., and T. L. Kay. 1989. Rendering fur with three dimensional textures. In Jeffrey Lane, ed., *Computer Graphics (SIGGRAPH '89 Proceedings),* 23:271–280.

Kajiya, J. T., and B. P. Von Herzen. 1984. Ray tracing volume densities. In H. Christiansen, ed., *Computer Graphics (SIGGRAPH '84 Proceedings),* 18:165–174.

Kalra, P., A. Mangili, N. Magnenat-Thalmann, and D. Thalmann. 1991. A multilayered facial animation system. In T. Kunii, ed., *Modeling in Computer Graphics.* New York: Springer-Verlag, 189–198.

Kameya, M., and J. C. Hart. 2000. Bresenham noise. *SIGGRAPH 2000 Conference Abstracts and Applications,* 182.

Karni, Z., and C. Gotsman. 2000. Spectral compression of mesh geometry. *Proceedings of SIGGRAPH 2000,* 279–286.

Karypis, G. 1999. Multi-level algorithms for multi-constraint hypergraph partitioning. Technical report #99–034, University of Minnesota.

Karypis, G., and V. Kumar. 1998. Multilevel algorithms for multi-constraint graph partitioning. *Proceedings of Supercomputing '98.*

Karypis, G., and V. Kumar. 1999. Multilevel *k*-way hypergraph partitioning. *Proceedings of IEEE Design Automation Conference,* 343–348.

Kautz, J., and M. D. McCool. 1999. Interactive rendering with arbitrary BRDFs using separable approximations. *Eurographics Rendering Workshop 1999,* 255–268.

Kay, T., and J. Kajiya. 1986. Ray tracing complex scenes. *Computer Graphics,* 20(4):269–278.

Kelley, K. W. 1988. *The home planet.* New York: Addison-Wesley.

Keppel, E. 1975. Approximating complex surfaces by triangulation of contour lines. *IBM Journal of Research and Development,* 19(1):2–11.

Kilgard, M. J. 2000. A practical and robust bump-mapping technique for today's GPUs. NVIDIA technical report.

Kim, T.-Y., and U. Neumann. 2001. Opacity shadow maps. *Proceedings of the Eurographics Rendering Workshop 2001.*

Kirk, D., and J. Arvo. 1991. Unbiased sampling techniques for image synthesis. *Computer Graphics,* 25(4):153–156.

Team LRN

Klassen, R. V. 1987. Modeling the effect of the atmosphere on light. *ACM Transactions on Graphics,* 6(3):215–237.

Kluyskens, T. 2002. Making good clouds. MAYA based QueenMaya magazine tutorial, *http://reality.sgi.com/tkluyskens aw/txt/tutor6.html.*

Kniss, J., G. Kindlmann, and C. Hansen. 2002. Multi-dimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics.* To appear.

Kniss, J., S. Premože, C. Hansen, and D. Ebert. 2002. Interactive translucent volume rendering and procedural modeling. Submitted to *IEEE Visualization 2002.*

Knuth, D. 1973. *The art of computer programming: Sorting and searching.* Reading, MA: Addison-Wesley.

Koza, J. R. 1992. *Genetic programming.* Cambridge, MA: MIT Press.

Lastra, A., S. Molnar, M. Olano, and Y. Wang. 1995. Real-time programmable shading. *1995 Symposium on Interactive 3D Graphics,* 59–66.

Lee, A. W. F., W. Sweldens, P. Schröder, L. Cowsar, and D. Dobkin. 1998. MAPS: Multiresolution adaptive parameterization of surfaces. *Proceedings of SIGGRAPH '98,* 95–104.

Lee, M. E., R. A. Redner, and S. P. Uselton. 1985. Statisically optimized sampling for distributed ray tracing. In B. A. Barsky, ed., *Computer Graphics (SIGGRAPH '85 Proceedings),* 19:61–67.

Leech, J. 1998. OpenGL extensions and restrictions for PixelFlow. Technical Report UNC-CH TR98–019, University of North Carolina at Chapel Hill, Dept. of Computer Science.

Levoy, M. 1988. Display of surfaces from volume data. *IEEE Computer Graphics and Applications,* 8(3):29–37.

Levoy, M. 1990a. Efficient ray tracing of volume data. *ACM Transactions on Graphics,* 9(3):245–261.

Levoy, M. 1990b. A hybrid ray tracer for rendering polygon and volume data. *IEEE Computer Graphics and Applications,* 10(2):33–40.

Levoy, M. 1990c. Volume rendering by adaptive refinement. *The Visual Computer,* 6(1):2–7.

Team LRN

Levy, B., and J. L. Mallet. 1998. Non-distorted texture mapping for sheared triangulated meshes. *Proceedings of SIGGRAPH '98,* 343–352.

Lewis, J. P. 1984. Texture synthesis for digital painting. In H. Christiansen, ed., *Computer Graphics (SIGGRAPH '84 Proceedings),* 18:245–252.

Lewis, J. P. 1986. Methods for stochastic spectral synthesis. In M. Green, ed., *Proceedings of Graphics Interface '86,* 173–179.

Lewis, J. P. 1987. Generalized stochastic subdivision. *ACM Transactions on Graphics,* 6(3):167–190.

Lewis, J. P. 1989. Algorithms for solid noise synthesis. In J. Lane, ed., *Computer Graphics (SIGGRAPH '89 Proceedings),* 23(3):263–270.

Lindenmayer, A. 1968. Mathematical models for cellular interactions in development. *Journal of Theoretical Biology,* 18:280–315.

Lokovic, T., and E. Veach. 2000. Deep shadow maps. In K. Akeley, ed., *Proceedings of ACM SIGGRAPH 2000, Computer Graphics Proceedings, Annual Conference Series,* 385–392.

Lorensen, W. L., and H. L. Cline. 1987. Marching cubes: A high resolution 3D surface construction algorithm. In M. C. Stone, ed., *Computer Graphics (SIGGRAPH '87 Proceedings),* 21:163–169.

Lorensen, W. L., and H. L. Cline. 1990. *Volume modeling.* First Conference on Visualization on Biomedical Computing.

Lovejoy, S., and B. B. Mandelbrot. 1985. Fractal properties of rain and a fractal model. *Tellus,* 37A:209–232.

Lynch, D. K. 1991. Step brightness changes of distant mountain ridges and their perception. *Applied Optics,* 30(24):308–313.

Ma, S., and H. Lin. 1988. Optimal texture mapping. *Proceedings of Eurographics '88,* 421–428.

Maillot, J., H. Yahia, and A. Verroust. 1993. Interactive texture mapping. *Proceedings of SIGGRAPH '93,* 27–34.

Mallat, S. G. 1989a. Multifrequency channel decompositions of images and wavelet modeling. *IEEE Transactions on Acoustic Speech and Signal Processing,* 37(12):2091–2110.

Team LRN

Mallat, S. G. 1989b. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 11:674–693.

Mandelbrot, B. B. 1982. *The fractal geometry of nature.* New York: W. H. Freeman.

Maruya, M. 1995. Generating a texture map from object-surface texture data. *Computer Graphics Forum* 14(3):C-397–C-403.

Max, N. L. 1986. Light diffusion through clouds and haze. *Computer Vision, Graphics and Image Processing,* 33(3):280–292.

Max, N. L. 1994. Efficient light propagation for multiple anisotropic volume scattering. *Fifth Eurographics Workshop on Rendering,* Darmstadt, Germany, 87–104.

McCool, M., and E. Fiume. 1992. Heirarchical poisson disk sampling distributions. *Proceedings of Graphics Interface '92,* 94–105.

McCool, M. C., and W. Heidrich. 1999. Texture shaders. *1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware,* 117–126.

Microsoft. 2001. Direct3D 8.0 specification. Available at *http://www.msdn.microsoft.com/directx*.

Milenkovic, V. J. 1998. Rotational polygon overlap minimization and compaction. *Computational Geometry: Theory and Applications,* 10:305–318.

Miller, G., and A. Pearce. 1989. Globular dynamics: A connected particle system for animating viscous fluids. *Computers and Graphics,* 13(3):305–309.

Miller, G. S. P. 1986. The definition and rendering of terrain maps. In D. C. Evans and R. J. Athay, eds., *Computer Graphics (SIGGRAPH '86 Proceedings),* 20(4):39–48.

Miller, G. S. P. 1988a. From wire-frames to furry animals. *Proceedings of Graphics Interface '88,* 135–145.

Miller, G. S. P. 1988b. The motion dynamics of snakes and worms. In J. Dill, ed., *Computer Graphics (SIGGRAPH '88 Proceedings),* 22:169–178.

Mine, A., and F. Neyret. 1999. Perlin textures in real time using OpenGL. Research report #3713, INRIA. *http://www-imagis.imag.fr/Membres/Fabrice.Neyret/publis/RR-3713-eng.html.*

Mitchell, D. 1987. Generating antialiased images at low sampling densities. *Computer Graphics,* 21(4):65–72.

Team LRN

Mitchell, D. 1991. Spectrally optimal sampling for distribution ray tracing. *Computer Graphics,* 25(4):157–164.

Mitchell, D. P., and A. N. Netravali. 1988. Reconstruction filters in computer graphics. In J. Dill, ed., *Computer Graphics (SIGGRAPH '88 Proceedings),* 22:221–228.

Miyata, 1990. A method of generating stone wall patterns. In F. Baskett, ed., *Computer Graphics (SIGGRAPH '90 Proceedings),* 24:387–394.

Molnar, S., J. Eyles, and J. Poulton. PixelFlow: High-speed rendering using image composition. *Computer Graphics,* 26(2):231–240.

Munkres, J. R. 1974. *Topology: A first course*. Englewood Cliffs, NJ: Prentice Hall.

Musgrave, F. K. 1988. Grid tracing: Fast ray tracing for height fields. Research Report, YALEU/DCS/RR 639. New Haven, CT: Yale University Dept. of Computer Science.

Musgrave, F. K. 1990. A note on ray tracing mirages. *IEEE Computer Graphics and Applications,* 10(6):10–12.

Musgrave, F. K. 1991. A random colormap animation algorithm. In J. Arvo, ed., *Graphics Gems II,* Boston: Academic Press.

Musgrave, F. K. 1993. *Methods for realistic landscape imaging*. New Haven, CT: Yale University.

Musgrave, F. K. 1994. *Methods for realistic landscape imaging*. Ph.D. thesis, Ann Arbor, Michigan, UMI Dissertation Services (Order Number 9415872).

Musgrave, F. K., C. E. Kolb, and R. S. Mace. 1989. The synthesis and rendering of eroded fractal terrains. In J. Eane, ed., *Computer Graphics (SIGGRAPH '89 Proceedings),* 23(3):41–50.

Musgrave, F. K., and B. B. Mandelbrot. 1989. Natura ex machina. *IEEE Computer Graphics and Applications,* 9(1):4–7.

Neyret, F. 1997. Qualitative simulation of convective cloud formation and evolution. *Eighth International Workshop on Computer Animation and Simulation,* Eurographics.

Nielson, G. M. 1991. Visualization in scientific and engineering computation. *IEEE Computer,* 24(9):58–66.

Nishimura, H., A. Hirai, T. Kawai, T. Kawata, I. Shirakawa, and K. Omura. 1985. Object modelling by distribution function and a method of image generation. *Journal of Papers Given at the Electronics Communication Conference '85,* J68-D(4). In Japanese.

Team LRN

Nishita, T., Y. Miyawaki, and E. Nakamae. 1987. A shading model for atmospheric scattering considering luminous intensity distribution of light sources. In M. C. Stone, ed., *Computer Graphics (SIGGRAPH '87 Proceedings),* 21:303–310.

Nishita, T., E. Nakamae, and Y. Dobashi. 1996. Display of clouds and snow taking into account multiple anisotropic scattering and sky light. In H. Rushmeier, ed., *SIGGRAPH '96 Conference Proceedings, Annual Conference Series,* Reading, MA: Addison-Wesley, 379–386.

Nishita, T., T. Sirai, K. Tadamura, and E. Nakamae. 1993. Display of the earth taking into account atmospheric scattering. *Computer Graphics, Annual Conference Series,* 175–182.

Norton, A., A. P. Rockwood, and P. T. Skolmoski. 1982. Clamping: a method of antialiasing textured surfaces by bandwidth limiting in object space. In *Computer Graphics (SIGGRAPH '82 Proceedings),* 16:1–8.

NVIDIA. 2001. Noise, component of the NVEffectsBrowser. Available at *http://www.nvidia.com/developer.*

Olano, M., and A. Lastra. 1998. A shading language on graphics hardware: The PixelFlow shading system. *Computer Graphics, Annual Conference Series (Proceedings of SIGGRAPH '98),* 159–168.

OpenGL ARB. 1999. *OpenGL programming guide,* third edition. Reading, MA: Addison-Wesley.

OpenGL Architecture Review Board. 2002. OpenGL extension registry. Available at *http://oss.sgi.com/projects/ogl-sample/registry/.*

Oppenheim, A. V., and R. W. Schafer. 1989. *Discrete-time signal processing.* Englewood Cliffs, NJ: Prentice Hall.

Owens, J. D., W. J. Dally, U. J. Kapasi, S. Rixner, P. Mattson, and B. Mowery. 2000. Polygon rendering on a stream architecture. *SIGGRAPH/Eurographics Workshop on Graphics Hardware,* 23–32.

Paglieroni, D. W., and S. M. Petersen. 1994. Height distributional distance transform methods for height field ray tracing. *ACM Transactions on Graphics,* 13(4):376–399.

Pallister, K. 2002. Generating procedural clouds in real time on 3D hardware. *http://cedar.intel.com/software/idap/media/pdf/games/procedural_clouds.pdf.*

Parish, Y. I. H., and P. Müller. 2001. Procedural modeling of cities. *Proceedings of SIGGRAPH 2001,* 301–308.

Team LRN

Park, N. 1993. *The wrong trousers*. Aardmann Animation.

Park, N. 1996. Personal communication.

Parke, F. 1982. Parameterized models for facial animation. *IEEE Computer Graphics and Applications,* 2(9):61–68.

Parke, F., and K. Waters. 1996. *Computer facial animation*. Wellesley, MA: A. K. Peters.

Peachey, D. R. 1985. Solid texturing of complex surfaces. In B. A. Barsky, ed., *Computer Graphics (SIGGRAPH '85 Proceedings),* 19:279–286.

Pedersen, H. K. 1995. Decorating implicit surfaces. In *Computer Graphics (SIGGRAPH '95 Proceedings),* 291–300.

Pedersen, H. K. 1996. A framework for interactive texturing operations on curved surfaces. *Proceedings of SIGGRAPH '96,* 295–302.

Peercy, M. S., J. Airey, and B. Cabral. 1997. Efficient bump mapping hardware. *Proceedings of SIGGRAPH '97, Computer Graphics Proceedings, Annual Conference Series,* 303–306.

Peercy, M. S., M. Olano, J. Airey, and J. Ungar. 2000. Interactive multipass programmable shading. *Computer Graphics, Annual Conference Series (Proceedings of SIGGRAPH 2000),* 425–432.

Peitgen, H. O., H. Jürgens, and D. Saupe. 1992. *Chaos and fractals: New frontiers of science*. New York: Springer-Verlag.

Peitgen, H. O., and D. Saupe, eds. 1988. *The science of fractal images*. New York: Springer-Verlag.

Perlin, K. 1985. An image synthesizer. In B. A. Barsky, ed., *Computer Graphics (SIGGRAPH '85 Proceedings),* 19(3):287–296.

Perlin, K. 1992. A hypertexture tutorial. *SIGGRAPH '92: Course Notes 23*.

Perlin, K. 1995. Real time responsive animation with personality. *IEEE Transactions on Visualization and Computer Graphics,* 1(1):5–15.

Perlin, K. 1997. Layered compositing of facial expression. *ACM SIGGRAPH '97 Technical Sketch,* 226–227

Perlin, K. 2002. Improving noise. *Computer Graphics,* 35(3).

Team LRN

Perlin, K., and A. Goldberg. 1996. Improv: A system for scripting interactive actors in virtual worlds. *Computer Graphics,* 30(3):205–216.

Perlin, K., and E. M. Hoffert. 1989. Hypertexture. In J. Lane, ed., *Computer Graphics (SIGGRAPH '89 Proceedings),* 23:253–262.

Perlin, K., and F. Neyret. 2001. Flow noise. *SIGGRAPH Technical Sketches and Applications*.

Phong, B. 1975. Illumination for computer generated pictures. *Communications of the ACM,* 18(6):311–317.

Piponi, Dan, and George Borshukov. 2000. Seamless texture mapping of subdivision surfaces by model pelting and texture blending. In *Computer Graphics (SIGGRAPH 2000 Proceedings),* 471–477.

Pixar. 1989. *The RenderMan interface: Version 3.1*. San Rafael, CA: Pixar.

Pixar. 1999. Future requirements for graphics hardware. Memo (April 12).

Pixar. 2000. *The RenderMan interface: Version 3.2*. San Rafael, CA: Pixar.

Plath, J. 2000. Realistic modelling of textiles using interacting particle systems. *Computers and Graphics,* 24(6):897–905.

Porter, T., and T. Duff. 1984. Compositing digital images. In H. Christiansen, ed., *Computer Graphics (SIGGRAPH '84 Proceedings),* 18:253–259.

Potmesil, M., and E. M. Hoffert. 1989. The pixel machine: A parallel image computer. *Computer Graphics,* 23(3):69–78.

Poulin, P., and A. Fournier. 1990. A model for anisotropic reflection. In F. Baskett, ed., *Computer Graphics (SIGGRAPH '90 Proceedings),* 24:273–282.

Praun, E., A. Finkelstein, and H. Hoppe. 2000. Lapped textures. In *Computer Graphics (SIGGRAPH 2000 Proceedings),* 465–470.

Press, W. H., B. P. FLannery, S. A. Tenkolsky, and W. T. Vetterting. 1986. *Numerical recipes*. New York: University of Cambridge.

Proudfoot, K., W. R. Mark, S. Tzvetkov, and P. Hanrahan. 2001. A real-time procedural shading system for programmable graphics hardware. *Computer Graphics, Annual Conference Series (Proceedings of SIGGRAPH 2001),* 159.

Team LRN

Prusinkiewicz, P., M. James, and R. Mech. 1994. Synthetic topiary. *Computer Graphics, Annual Conference Series,* 351–358.

Prusinkiewicz, P., and A. Lindenmayer. 1990. *The algorithmic beauty of plants.* New York: Springer-Verlag.

Purcell, T. J., I. Buck, W. R. Mark, and P. Hanrahan. 2002. Ray tracing on programmable graphics hardware. *Computer Graphics, Annual Conference Series (Proceedings of SIGGRAPH 2002),* 703.

Reeves, W. T. 1983. Particle systems: A technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics,* 2:91–108.

Reeves, W. T., and R. Blau. 1985. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *Computer Graphics* 19:313–322.

Reeves, W. T., D. H. Salesin, and R. L. Cook. 1987. Rendering antialiased shadows with depth maps. In M. C. Stone, ed., *Computer Graphics (SIGGRAPH '87 Proceedings),* 21:283–291.

Reynolds, C. W. 1987. Flocks, herds, and schools: A distributed behavioral model. In M. C. Stone, ed., *Computer Graphics (SIGGRAPH '87 Proceedings),* 21:25–34.

Rhoades, J., G. Turk, A. Bell, U. Neumann, and A. Varshney. 1992. Real-time procedural textures. *1992 Symposium on Interactive 3D Graphics,* 25(2):95–100.

Rich, E. 1983. *Artificial intelligence.* New York: McGraw-Hill.

Rioux, M., M. Soucy, and G. Godin. 1996. A texture-mapping approach for the compression of colored 3D triangulations. *Visual Computer,* 12(10):503–514.

Rixner, S., W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. Owens. 1998. A bandwidth-efficient architecture for media processing. *Proceedings of the 31st Annual International Symposium on Microarchitecture,* 3–13.

Rubin, S. M., and T. Whitted. 1980. A 3-dimensional representation for fast rendering of complex scenes. *Proceedings of SIGGRAPH '80,* 110–116.

Rushmeier, H. E., and K. E. Torrance. 1987. The zonal method for calculating light intensities in the presence of a participating medium. In M. C. Stone, ed., *Computer Graphics (SIGGRAPH '87 Proceedings),* 21:293–302.

Ruskai, M. B., ed. 1992. *Wavelets and their applications.* Boston: Jones and Bartlett.

Team LRN

Sabella, P. 1988. A rendering algorithm for visualizing 3D scalar fields. In J. Dill, *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22:51–58.

Sakas, G. 1993. Modeling and animating turbulent gaseous phenomena using spectral synthesis. *The Visual Computer,* 9(4):200–212.

Salisbury, M. B., S. Anderson, R. Barzel, and D. Salesin. 1994. Interactive pen-and-ink illustration. *Computer Graphics Proceedings, Annual Conference Series*, 101–108.

Samek, M. 1986. Texture mapping and distortion in digital graphics. *The Visual Computer,* 2(5):313–320.

Sander, P. V., J. Snyder, S. J. Gortley, and H. Hoppe. 2001. Texture mapping progressive meshes. *Proceedings of SIGGRAPH 2001,* 409–416.

Saupe, D. 1988. Algorithms for random fractals. In H. O. Peitgen and D. Saupe, eds., *The science of fractal images,* New York: Springer-Verlag, 71–136.

Saupe, D. 1989. Point evaluation of multi-variable random fractals. In H. Juergen and D. Saupe, eds., *Visualisierung in Mathematik und Naturissenschaft-Bremer Computergraphik Tage 1988,* Heidelberg: Springer-Verlag.

Saupe, D. 1992. Random fractals in image synthesis. In P. Prusinkiewicz, ed., *Fractals: From Folk Art to Hyperreality, SIGGRAPH '92 Course Notes 12*.

Sayre, R. 1992. Antialiasing techniques. In A. Apodaca and D. Peachey, eds., *Writing RenderMan Shaders, SIGGRAPH '92 Course Notes 21,* 109–141.

Schacter, B. J. 1980. Long-crested wave models. *Computer Graphics and Image Processing,* 12:187–201.

Schacter, B. J., and N. Ahuja. 1979. Random pattern generation processes. *Computer Graphics and Image Processing,* 10:95–114.

Schlick, C. 1994. Fast alternatives to Perlin's bias and gain functions. In P. S. Heckbert, ed., *Graphics Gems* IV, volume IV, 401–403, Cambridge, MA: Academic Press Professional.

Schroeder, W. J., J. A. Zarge, and W. E. Lorensen. 1992. Decimation of triangle meshes. In E. E. Catmull, ed., *Computer Graphics (SIGGRAPH '92 Proceedings),* 26(2):65–70.

Segal, M., and K. Akeley. 2001. The OpenGL graphics system: A specification, version 1.2.1. Available at *http://www.opengl.org/.*

Team LRN

Shannon, S. 1995. The chrome age: Dawn of virtual reality. *Leonardo,* 28(5):369–380.

Shirley, P. 1993. Monte Carlo simulation. *SIGGRAPH '92 Course Notes 18*.

Shoemake, K. 1991. Interval sampling. In J. Arvo, ed., *Graphics Gems II,* Boston: Academic Press, 394–395.

Sims, K. 1990. Particle animation and rendering using data parallel computation. In F. Baskett, ed., *Computer Graphics (SIGGRAPH '90 Proceedings),* 24:405–413.

Sims, K. 1991a. Artificial evolution for computer graphics. In T. W. Sederberg, ed., *Computer Graphics (SIGGRAPH '91 Proceedings),* 25(4):319–328.

Sims, K. 1991b. Interactive evolution of dynamical systems. *Proceedings of the European Conference on Artificial Life*. Paris: MIT Press.

Sims, K. 1994. Evolving virtual creatures. In Andrew Glassner, ed., *Computer Graphics (SIGGRAPH '94 Proceedings),* 15–22.

Skinner, R., and C. E. Kolb. 1991. Noise.c component of the Rayshade ray tracer.

Smith, A. R. 1984. Plants, fractals and formal languages. In H. Christiansen, ed., *Computer Graphics (SIGGRAPH '84 Proceedings),* 18:1–10.

Snyder, J. M., and A. H. Barr. 1987. Ray tracing complex models containing surface tessellations. *Computer Graphics* 21(4):119–128.

Stam, J. 1995. Multiple scattering as a diffusion process. *Eurographics Rendering Workshop*.

Stam, J. 1999. Stable fluids. In A. Rockwood, ed., *Proceedings of SIGGRAPH '99, Computer Graphics Proceedings, Annual Conference Series,* 121–128.

Stam, J., and E. Fiume. 1991. A multiple-scale stochastic modelling primitive. *Proceedings of Graphics Interface '91,* 24–31.

Stam, J., and E. Fiume. 1993. Turbulent wind fields for gaseous phenomena. In J. T. Kajiya, ed., *Computer Graphics (SIGGRAPH '93 Proceedings),* 27:369–376.

Stam, J., and E. Fiume. 1995. Depicting fire and other gaseous phenomena using diffusion processes. In R. Cook, ed., *SIGGRAPH '95 Conference Proceedings, Annual Conference Series,* Reading, MA: Addison-Wesley, 129–136.

Team LRN

Standler, B., and J. Hart. 1994. A Lipschitz method for accelerated volume rendering. *1994 Symposium on Volume Visualization.*

Stephenson, N. 1992. *Snow crash.* New York: Bantam Doubleday.

Sutherland, I. E. 1963. Sketchpad: A man-machine graphical communication system. *Proceedings of Spring Joint Computer Conference.*

Szeliski, R., and D. Tonnesen. 1992. Surface modeling with oriented particle systems. *Computer Graphics (Proceedings of SIGGRAPH '92),* 26(2):185–194.

Tadamura, K., E. Nakamae, K. Kaneda, M. Baba, H. Yamashita, and T. Nishita. 1993. Modeling and skylight and rendering of outdoor scenes. *Eurographics '93,* 12(3):189–200.

Thompson, D. 1942. *On growth and form.* Cambridge: University Press. Abridged edition (1961) edited by J. T. Bonner.

Thorne, C. 1997. Convert solid texture. Software component of Alias|Wavefront Maya 1.

3Dlabs. 2001. *OpenGL 2.0 shading language white paper, version 1.1.*

Todd, S., and W. Latham. 1993. *Evolutionary art and computers.* Boston, MA: Academic Press.

Tricker, R. 1970. *The science of the clouds.* Amsterdam: Elsevier.

Turk, G. 1991. Generating texures on arbitrary surfaces using reaction-diffusion. In T. W. Sederberg, ed., *Computer Graphics (SIGGRAPH '91 Proceedings),* 25:289–298.

Turk, G. 2001. Texture synthesis on surfaces. In *Computer Graphics (SIGGRAPH '01 Proceedings),* 347–354.

University of Illinois Department of Atmospheric Sciences. 2002. Cloud catalog. *http://ww2010.atmos.uiuc.edu/(Gh)/guides/mtr/cld/cldtyp/home.rxml.*

Upstill, S. 1990. *The RenderMan companion.* Reading, MA: Addison-Wesley.

Van Gelder, A., and K. Kim. 1996. Direct volume rendering with shading via 3D textures. *ACM/IEEE Symposium on Volume Visualization,* 22–30.

van Wijk, J. J. 1991. Spot noise-texture synthesis for data visualization. In T. W. Sederberg, ed., *Computer Graphics (SIGGRAPH '91 Proceedings),* 25:309–318.

Team LRN

Velho, L., K. Perlin, L. Ying, and H. Biermann. 2001. Procedural shape synthesis on subdivision surfaces. *Proc. Symposium on Computer Graphics and Image Processing (SIBGRAPI),* 146–153.

Voorhies, D. 1991. Space filling curves and a measure of coherence. In J. Arvo, ed., *Graphics Gems II,* Boston: Academic Press, 26–30.

Voss, R. 1983. Fourier synthesis of Gaussian fractals: Noises, landscapes, and flakes. *SIGGRAPH '83: Tutorial on State of the Art Image Synthesis,* 10.

Voss, R. F. 1988. Fractals in nature: From characterization to simulation. In H. O. Peitgen and D. Saupe, eds., *The science of fractal images,* New York: Springer-Verlag, 21–70.

Wann Jensen, H., and P. H. Christensen. 1998. Efficient simulation of light transport in scenes with participating media using photon maps. In M. F. Cohen, ed., *Proceedings of SIGGRAPH 98, Computer Graphics Proceedings, Annual Conference Series,* 311–320.

Ward, G. 1991. A recursive implementation of the Perlin noise function. In J. Arvo, ed., *Graphics Gems II,* 396–401. Academic Press Professional.

Waters, K. 1987. A muscle model for animating three-dimensional facial expression. In M. C. Stone, ed., *Computer Graphics (SIGGRAPH '87 Proceedings),* 21:17–24.

Wei, L.-Y., and M. Levoy. 2000. Texture synthesis using tree-structured vector quantization. In *Computer Graphics (SIGGRAPH 2000 Proceedings),* 479–488.

Wei, L.-Y., and M. Levoy. 2001. Texture synthesis over arbitrary manifold surfaces. In *Computer Graphis (SIGGRAPH '01 Proceedings),* 355–360.

Westover, L. 1990. Footprint evaluation for volume rendering. In F. Baskett, ed., *Computer Graphics (SIGGRAPH '90 Proceedings),* 24:367–376.

Williams, L. 1983. Pyramidal parametrics. *Computer Graphics,* 17(3):1–11.

Willis, P. J. 1987. Visual simulation of atmospheric haze. *Computer Graphics Forum,* 6(1):35–42.

Winston, P. H., and B. K. P. Horn. 1984. *LISP,* 2nd edition. Reading, MA: Addison-Wesley.

Witkin, A., and P. Heckbert. 1994. Using particles to sample and control implicit surfaces. In Andrew Glassner, ed., *Computer Graphics (SIGGRAPH '94 Proceedings),* 269–278.

Witkin, A., and M. Kass. 1991. Reaction-diffusion textures. In T. W. Sederberg, ed., *Computer Graphics (SIGGRAPH '91 Proceedings),* 25:299–308.

Team LRN

Worley, S. 1993. Practical texture implementation. *Procedural Modeling and Rendering Techniques Course Notes, ACM SIGGRAPH '93,* vol. 12.

Worley, S. 1996. A cellular texture basis function. *Proceedings of SIGGRAPH '96*, 291.

Worley, S., and J. Hart. 1996. Hyper-rendering of hyper-textured surfaces. *Implicit Surfaces '96*.

Wyvill, B., and J. Bloomenthal. 1990. Modeling and animating with implicit surfaces. *SIGGRAPH '90: Course Notes 23*.

Wyvill, B., C. McPheeters, and G. Wyvill. 1986. Data structure for soft objects. *The Visual Computer,* 2(4):227–234.

Wyvill G., B. Wyvill, and C. McPheeters. 1987. Solid texturing of soft objects. *IEEE Computer Graphics and Applications,* 7(4):20–26.

Yang, X. D. 1988. Fuzzy disk modeling and rendering of textured complex 3D surfaces of real objects. Technical Report, TR-88–414. New York: New York University.

Team LRN

INDEX

Symbols and Numbers
2D antialiasing, 176
2D slices of noise functions, 74
2D texture mapping
  for clouds, 267
  limitations of, 10
  methods, 197–199
3D Nature's World Construction Set, 580
3D tables for controlling animation, 244–254
  accessing table entries, 245
  breeze effect using attractor, 247, 248, 252–253, 254
  combinations of functions, 251–254
  common dimensions for, 245
  flow into hole in wall, 253–254, 255
  functional flow field functions, 246–251
  functional flow field tables, 245, 246
  nonuniform spacing between entries, 244–245
  overview, 244–245
  vector field tables, 245
  wind effects, 252–254
3D textures, converting to image maps, 197–199
4D cellular texturing, 149
4D noise functions, 84, 181

`abs` function (RenderMan), 29–30
abstraction, 2
a-buffer rendering algorithm, 205
additive cascades, multiplicative cascades vs., 440
Advanced RenderMan, 102, 129
Adventures of André & Wally B, The, 259, 260
aerial perspective, 529–530
aesthetic *n*-space model, 548–549, 559
air currents. *See* wind
Air Force One, 473, 525
algorithms
  a-buffer rendering algorithm, 205
  for cumulus clouds, 272, 274
  explicit noise algorithms, 82–83
  genetic algorithms vs. genetic programming, 558–559
  multipass pixel shader implementation of noise function, 423–425
  QAEB, 513–514
  real-time procedural solid texturing, 413–416
  REYES (PRMan), 103–106
  ridge algorithm for bump mapping, 187
  shadow table algorithm, 207–208
  volume-rendering algorithm for gases, 206
Alias Dynamation, 524
aliasing. *See also* antialiasing
  brick texture example, 56, 57
  cyberspace and, 622
  defined, 52
  fractal octaves and, 585
  image textures vs. procedural textures and, 15
  index aliasing, 158–166
  maximum frequency and, 53, 54
  MojoWorld function fractals and, 602–604
  Nyquist frequency and, 53–54
  planetary rings example, 163–166
  in procedural textures, 55–56
  sampling rate and, 53, 54
  in signal processing, 53–54
  signal processing concepts, 52–56
  `smoothstep` function for avoiding, 31
  sources of, 158–160
  supersampling and, 54
  temporal, 175
  white noise and, 68
Alias's Maya animation system. *See* Maya animation system
`ambient` function (RenderMan), 21
amplification
  data amplification algorithms for procedural geometry, 305, 312–314
  database amplification, 2

Team LRN

amplification (*continued*)
by fractals, 436
intermediate representation and, 313
proceduralism and, 549
amplitude of fractals, 433
analytic prefiltering
box filter for, 61–62, 64–65
brick texture example, 64–66
filters with negative lobes, 62
integrals for, 3–4, 65
overview, 56–57
`smoothstep` function for, 62
summed-area table method, 63–64
Animatek's World Builder, 580
animating gaseous volumes, 235–243
helical path effects, 236–243
rolling fog, 238–239
smoke rising, 239–243
steam rising from teacup, 236–238
animating hypertextures, 254–256
animating solid spaces, 227–261
animation paths, 228–229
approaches, 227–228
changing solid space over time, 227, 229–232
ease-in and ease-out procedures, 229
hypertextures, 254–256
marble procedure for examples, 229
moving screen space point through solid space, 227, 228, 232–233, 235–243
solid textured transparency, 233–235
three-dimensional tables for controlling animation, 244–254
animating volumetric procedural clouds, 279–283
implicit primitive animation, 280–283
procedural animation, 279–280
animation
aliasing problems magnified by, 157
cellular texturing for, 149
clip animation, 391
ease-in and ease-out procedures, 229
flame shader, noise-based procedural, 124–129
paths for, 228–229
of real-time effects, 291
textural limb animation, 387, 391–395
texture for facial movement, 395–409
textures for, 196–197
time-animated fractal noise, 181
`anisotropic1` shader, 114–115
`anisotropic2` shader, 115–116
anisotropic shading models
development of, 8
Heidrich/Banks model, 112–116
antialiased rendering of procedural textures, 369–376
background, 370
basic idea, 370–372
detailed description, 372–373
examples, 374–376
high-contrast filter, 371–372, 373–374
ifpos pseudofunction for, 371, 372–373
nested conditionals and, 369
overview, 369, 376
antialiasing, 52–67, 157–176, 369–376. *See also* aliasing
alternatives to low-pass filtering, 66–67
analytic prefiltering techniques, 56–57, 61–66
blending between texture versions for, 66
blur effects for, 175–176
blurring color map for, 175
box filter for, 61–62, 64–65
brick texture example, 64–66
bump mapping and, 59
clamping method, 56, 59–61
edges that result from conditionals, 369–376
efficiency issues, 157
emergency alternatives, 175–176
exaggerating spot size for, 175
filters with negative lobes for, 62
fractal octave reduction for, 585
high-contrast filter for, 371–372, 373–374
`if` functions vs. `step` functions for, 28
image textures vs. procedural textures and, 15
importance of, 157
index antialiasing, 160–166
integrals for, 63–64, 65
low-pass filtering, 54
methods for procedural textures, 56–67
modeling input distribution, 161–162
object space vs. screen space shading and, 106
optimization and verification, 173–175
PhotoRealistic RenderMan scheme, 55

planetary rings example, 163–166
in QAEB tracing, 516
reference image for, 174
by renderers, 55–56
rendering procedural textures, 369–376
rendering texture as 2D image, 176
sampling and bumping, 170–173
sampling rate determination, 57–59
`smoothstep` function for, 62
spot size calculation, 166–170
stochastic sampling, 55, 67
sum table for, 162–163
summed-area table method, 63–64
supersampling, 10–13, 54, 66–67, 157
temporal, 175
test scene design, 174
by texture instead of renderer, 170–173
two-dimensional, 176
API routines
interface between shaders and applications, 118–121
OpenGL, 118
in Stanford shading language, 118–120
Apollo 13, 473
`applytexture` program, 111–112
architectexture, 359–362
area-weighted mesh atlas, 417–418
art, computer's role in, 482
artifacts
frequency multiplier and lattice artifacts, 88–89
grid-oriented artifacts, reducing, 348
from noise algorithms, 88–89, 180–181, 348
rotation matrices for hiding, 180, 181
seam artifacts, avoiding, 419–421
terrain creases as, 498
ArtMatic, 557
assembly language, for shading, high-level language vs., 100–101
atlas
area-weighted mesh atlas, 417–418
based on clusters of proximate triangles, 418, 419
defined, 416
length-weighted atlas, 418
for real-time procedural solid texturing, 416–419
to support MIP mapping, 418–419
uniform meshed atlas, 416–417
atmosphere, fractal properties of, 570
atmospheric dispersion effects for gaseous phenomena, 205
atmospheric models, 529–544. *See also* color perspective; GADDs (geometric atmospheric density distribution models)
aerial perspective, 529–530
Beer's law and homogenous fog, 531–532
curved, 569–570
dispersion effects for gaseous phenomena, 205
elements involved in, 529, 530
exponential mist, 532, 534
extinction, 531–532
global context for landscape, 530
integration schemes for, 529, 534–544
minimal Rayleigh scattering approximation, 536, 539
numerical quadrature with bounded error for radial GADDs, 542–544
optical depth, 531
optical paths, 531, 534
physical models vs., 530
radially symmetric planetary atmosphere, 534–536
scattering models for, 529, 530, 536, 539
terminology, 531
trapezoidal quadrature for radial GADDs, 539–542
atmospheric perspective, 529
attractors
combinations of functions, 251–254
creating repulsors from, 246
extensions of spherical attractors, 249
overview, 247
spherical, 247–249
Avatars, 391
axioms in L-system, 308

banded color, sine function for, 230
band-limited fractals, 434
bandwidth
memory bandwidth and performance tuning, 110
in signal processing, 52
`basic_gas` function
overview, 214–215
patchy fog using, 216
power function effects on, 215–216
sine function effects on, 215, 216
steam rising from teacup using, 216–219, 220

basis functions of fractals
defined, 582
manipulating for variety, 450
MojoWorld, 450, 582, 583, 590–597
overview, 432–433
for random fractals, 583
visual effects for homogenous fBm terrain models, 497–498
basketball, texture baking example for, 198–199
Beer's law, atmospheric models and, 531–532
bevel for bump mapping. *See* ridge for bump mapping
`bias` function
`gain` function and, 39
for gamma correction, 37–38
$\texttt{bias}_b$ function
defined, 339
for tuning fuzzy regions, 339, 340
bicubic patch modeling with L-system, 311
bilinear filtering, seam artifacts and, 420–421
billboards
for cloud modeling, 268
for gases, 209
billowing clouds, 523–524
binding
early vs. late, performance and, 117–118
of shaders, program responsibility for, 120
Stanford shading language early-binding model, 118
black-body radiators, 461
black-box textures, future of, 200–201
blending capability of GPUs, 290
blinking, 405. *See also* texture for facial movement
blobby molecules, 3
Blue Moon Rendering Tools (BRMT), 102
blurring
blur effects for antialiasing, 175–176
color map for antialiasing, 175
bombing, random placement patterns based on, 91–94
bounding volumes for procedural geometry, 330–332
box filter for antialiasing, 61–62, 64–65
`boxstep` function (RenderMan)
analytic prefiltering using, 61–62
brick texture antialiasing example, 64–65
brick textures
aliasing in, 56, 57
antialiased example, 64–66
bump-mapped brick, 41–46
bump-mapped mortar, 44–45
determining whether point is in brick or mortar, 41
images rendered using, 42, 46, 66
offsetting alternate rows, 41
perturbed, 89–90
procedural texture generator, 39–41
pulses generating brick shape, 41
`scoord` and `tcoord` texture coordinates, 40
BRMT (Blue Moon Rendering Tools), 102
Bryce software, 580
bump mapping. *See also* displacement mapping
adding to basic texture, 183
antialiasing and, 59, 170–173
bevel for three-dimensional appearance, 183, 184
brick mortar example, 44–45
brick texture example, 41–46
cellular textures, 140, 141, 142
correct vs. incorrect, 171–173
geometry of, 42
impressive textures' use of, 183
methods, 183–187
overview, 9, 41–43
ridge algorithm for, 183–187
rotation matrices for hiding artifacts, 180
texture spaces and, 45–46
for water ripples, 461, 463
"Bursting," 17
bushes, L-system for modeling, 318–320

C language, translating RenderMan shading language into, 19
`calc_noise` function, 212–213
`calculatenormal` function, described, 44
`calc_vortex` procedure, 250–251
`camera` space (RenderMan)
`current` space and, 24
solid textures and, 24–25
Carefree Gum, "Bursting," 17
Carmack, John, 130
cascades, multiplicative vs. additive, 440
`ceil` function (RenderMan), 33
`ceilf` function (ANSI C implementation), macro alternative to, 33
cellular texturing, 135–155
2D and 4D variants, 149
basis functions, 136–140
density of feature points in, 144–145, 146–147, 149
distance metric for, 147–149

extensions and alternatives, 147–149
feature points defined, 136
fractal combinations, 138–140, 141
implementation strategy, 140–149
isotropic property of, 142, 143, 145–147
linear combinations of functions *F*, 137–138, 139
"Manhattan" distance metric, 148
mapping function values onto colors and bumps, 137
modifying the algorithm, 136, 140–141
neighbor testing, 144–145
noise compared to, 135–136
nonlinear combinations of polynomial products, 140
partitioning space into cellular regions, 136–137, 142–144
Poisson distribution of feature points, 145–147
population table for, 145–147
properties of functions *F*, 137
random number generator for, 143
sample code, 149–155
speed vs. isotropy in, 146–147
Cg language (NVIDIA), 111, 292–296
checkerboard basis function (MojoWorld), 596–597
checkerboard pattern generation, 39
cirrus cloud models, 275–276, 277, 282, 283, 453–458
`Cirrus` procedure, 275–276
`clamp` function (RenderMan), 28–29
clamping
closeness to texture pattern and, 60–61
described, 56
fade-out for sampling rate changes, 60
limitations of, 61
spectral synthesis antialiasing using, 59–61, 86–87
for `turbulence` function, 86–87
clip animation, 391
clipping planes in QAEB tracing, 514–515
cloth, hypertexture example, 357, 359, 360, 361
`cloudplane` shader, 50
clouds. *See also* fog; gases; volumetric cloud modeling and rendering
altitude and types of, 266
animating volumetric procedural clouds, 279–283
atmospheric dispersion effects, 267
basics, 263, 266–267
billowing, 523–524
challenge of, 448
cirrus cloud models, 275–276, 277, 282, 283, 453–458
cloud creatures, 273, 278
commercial packages for rendering, 284, 285
Coriolis effect, 458–460
cumulus cloud models, 272–274, 282
difficulties modeling, 263, 267
distorted noise function for, 450–453
ellipsoid surfaces for modeling, 195, 267
example photographs, 264–265
fBm-valued distortion for, 454–456
forces shaping, 263, 266
fractal solid textures for, 448–460
functional composition for modeling, 453–454
hardware acceleration for, 298–301
illumination models, 266–267
for interactive applications, 267–268
interactive models, 283–284, 285
jet stream warping, 279–280
MojoWorld for, 611
ontogenetic modeling of hurricane, 456–458
previous approaches to modeling, 267–268
psychedelic, 525, 526
puffy, 448–449
QAEB-traced hypertextures for, 520–525, 526
real-time, 298–301
spectral synthesis for simulating, 50–51
stratus cloud models, 276, 277–278, 282, 283
surface-based modeling approaches, 267–268
on Venus, 458–460
visual characteristics, 266
volume perturbation for modeling, 299–301
volumetric modeling, 268–272
volumetric rendering, 272–279
weather forecasting and, 266
color. *See also* RGB color
black-body radiators, 461
blurring color map for antialiasing, 175
color mappings, 181–182
color splines, 182, 189–192

Team LRN

color (*continued*)
color table equalization, 189, 190–192
fBm coloring, 477–478
for fire, 461
GIT texturing system, 478–480
index aliasing, 158–163
layering texture patterns, 25–26
L-system specification, 311–312
for marble using NVIDIA's Cg language, 293–294
"multicolor" texture, 482–485
normalizing noise levels for, 189–192
`plastic` shader output, 21
random coloring methods, 477–485
sine function for banding, 230
solid color texture, 196
`spline` function for mapping, 36
usage in this book, 182
color perspective. *See also* atmospheric models
in *Carolina,* 534
defined, 529
in *Himalayas,* 537
in planetary atmosphere, 538
raw form, 538
Rayleigh scattering and, 536
color splines
color table equalization, 189, 190–192
normalizing noise levels, 189–192
overview, 182
color table equalization
histogram of noise values, 190, 191
mapping percentage levels to noise values, 190–192
need for, 189, 192
normalization plot, 190, 191
complexity
fractal vs. nonfractal, 431
fractals and visual complexity, 429, 506, 567–571
hyperspace in MojoWorld and, 588–589
realism and, 434–435
as work, 434
composition. *See* functional composition
computational precision issues for GPUs, 288
`compute_color` function for raymarcher renderer, 350, 351
computer graphics research community, 576, 578–579
"Computer Rendering of Stochastic Models," 576
computer's role in art, 482
`compute_shading` function for raymarcher renderer, 350
conditional functions
`abs` (RenderMan), 30
antialised rendering of edges resulting from, 369–376
`clamp` (RenderMan), 28–29
`max` (RenderMan), 28–29
`min` (RenderMan), 28–29
`smoothstep` (RenderMan), 30–31
`step` (RenderMan), 27–28
conferencing, electronic, 391
conservation of angular momentum, spiral vortex functions based on, 251
constant density medium for modeling gases, 204
convective cloud modeling, 268
convolution kernel, 595
convolution noise
lattice, 78–80
sparse, 80–82
coordinate systems in RenderMan, 24. *See also* texture spaces
Coriolis effect, 458–460
`cosf` function (ANSI C implementation), RenderMan `cos` function vs., 31
cosine function
graph of, 32
overview, 31
for rotation on helical path, 228
CPUs
GPUs vs., 100, 102, 109
migrating procedural techniques to GPUs from, 287–289
real-time procedural solid texturing on, 421
virtualization supported by, 109
crop circles, PGI for modeling, 326
crossover scales
of fractals, 434
in MojoWorld, 606–607
cubic Catmull-Rom spline interpolation for value noise, 71–72
cumulus cloud models, 272–274, 282
`cumulus` procedure, 272, 274
current space, described, 24
cyberspace
aliasing and, 622
as context for information content, 621, 622, 623
defined, 617
features of "natural" cyberspace, 619–620
fractal geometry of, 620–622
MojoWorld photorealistic renderer and, 617–618

social aspects of, 621–622
vision of, 618–620
cyclic scene graphs, 320, 321
`cyclone` procedure, 456–458

dance
bias and gain controls for, 397
emotional gestures in, 391
Dark Tree, 447
data amplification algorithms
for procedural geometry, 312–314
described, 305
intermediate representation and, 313
in L-system, 312–314
for particle systems, 313–314
data types, GPU support for, 108–109
database amplification, 2
"Death Star" surface, 148
debugging, image textures vs. procedural textures and, 14
declarative components of procedural techniques, 12
deep shadow map for gases, 208
degrees of freedom in *n*-spaces, 548
DEM (digital elevation map) format, 494
density
constant, for modeling gases, 204
of feature points in cellular texturing, 144–145, 146–147, 149
function for hypertexture explosion example, 355
height attenuation for rising steam, 218–219
height attenuation for smoke dispersion, 220
implicit density functions for cloud modeling, 270–271
power function and gas density, 215–216
raymarcher renderer user-defined functions, 350–352
scaling vector returned by functional flow field functions, 246–247
smoke density function, 367–368
spherical attenuation to confine steam within cup radius, 216–218
turbulence-based, for cloud modeling, 271
`density_function` for raymarcher renderer, 350
depth buffer renderers, implicit vs. explicit procedures in, 13
depth-of-field effects, object space vs. screen space shading and, 105–106
derivative discontinuities in `turbulence` function, 86
`diffuse` function (RenderMan), 21
diffuse model, 7–8
digital elevation map (DEM) format, 494
digital terrain elevation data (DTED) format, 494
dilation symmetry
in fractal geometry, 621
fractals as, 431, 573–574
dimension, 588
Direct3D, OpenGL vs., 102
displacement mapping. *See also* height fields
in MojoWorld, 610–611
object space vs. screen space shading and, 104, 105–106
overview, 9
scanline rendering vs. ray tracing and, 511
for water ripples, 461–463
distorted fractal functions in MojoWorld, 606
distorted noise (`DistNoise`) function, 450–453
for altering fractal basis function, 450
C++ version, 450
RenderMan version, 452–453
terse version, 452
undistorted noise function vs., 451–452
`VecNoise` function in, 450–451
domain distortion
for cirrus cloud modeling, 453–454
`DistNoise` function based on, 450–453
in MojoWorld, 604–606
domain function dimensions in MojoWorld, 587–588
dot product capability of GPUs, 290
Dr. Mutatis program
demise of, 550
images generated by, 558–563
DTED (digital terrain elevation data) format, 494
`Du` and `Dv` functions (RenderMan), 58–59
`du` and `dv` variables (RenderMan), 58
dynamic bounding volumes, 331

earth textures. *See also* MojoWorld; procedural fractal terrains
fractal solid textures for earth textures, 466–477
Gaea (Earth-like planet), 467–472

earth textures (*continued*)
sedimentary rock strata, 466–467, 468
Selene (Moon), 473–477
ease-in and ease-out procedures, 229
edge events, 370. *See also* antialiased rendering of procedural textures
egg shape
for creating hypertextures, 354
explosion inside, 355, 356, 357
electronic conferencing, 391
emittance, volume rendering with surface textures and, 196
emotive gesturing. *See* textural limb animation
`environment` function (RenderMan), 22
environment mapping, 9
E-on Software's Vue d'Esprit, 580
Euclidean geometry, fractal geometry vs., 620–621
exact self-similarity, statistical self-similarity vs., 435, 575
explicit procedures
defined, 12
implicit procedures vs., 12–14
explosions
hypertexture example, 354–356
inside an egg, 355, 356, 357
pyroclastic flow, 523–524
expression trees, 553–556
extinction in atmospheric models, 531–532

`F` (turtle graphics symbol), 308
`fabs` function with turbulence functions, 369
`faceforward` function (RenderMan), 21
Facial Action Coding System (FACS), 396
facial expressions. *See* texture for facial movement
fade in, solid color texture for, 196
fast Fourier transform (FFT), spectral synthesis and, 49
fBm. *See* fractional Brownian motion (fBm)
feature points in cellular texturing
defined, 136
density of, 144–145, 146–147, 149
isotropic distribution of, 142, 143, 145–147
precomputed, 149
testing regions of space for, 142–145
feature spaces for textures
overview, 25
star texture, 46
FFT (fast Fourier transform), spectral synthesis and, 49
filtering capability of GPUs, 290
fire. *See also* volumetric procedural modeling and texturing
colors for, 461
fBm-valued distortion for, 460–461, 462
fractal solid textures for, 460–461
noise-based procedural flame shader, 124–129
particle system for wall of fire, 257, 258
pyroclastic flow, 523–524
QAEB-rendered fireballs, 525
flagstone texture, 140, 142
flame. *See* fire
`flame` procedure, 460–461
flexibility
GPU issues, 288–289
procedural techniques and, 2
`floor` function (RenderMan), 32–33
`floorf` function (ANSI C implementation), macro alternative to, 33
flow field functions. *See* functional flow field functions
flow noise, 384–387
example flow textures, 387, 388–389
need for, 384–385
pseudoadvection in, 386–387
rotation gradients for, 385–386
fog. *See also* gases; volumetric procedural modeling and texturing
animating solid textured transparency, 233–235
Beer's law and homogenous fog, 531–532
exponential mist, 532, 534
hardware acceleration for space-filling fog, 297–298
patchy fog modeling, 216
rolling fog animation, 238–240
`fog` procedure, 233–234
described, 233
helical path for, 233
values for, 235
Fourier synthesis
basis function for, 497
for cloud modeling, 267
for controlling transparency, 204
point evaluation vs., 490
Fourier transform
domain change from, 49
inverse, 49

signal processing and, 52–53
for spectral synthesis, 49–51
`frac` macro, 65
fractal dimension
amplitude scaling and, 89
defined, 582–583
for homogenous fBm terrain models, 495–497
overview, 432–433
for random fractals, 583–584
roughness of surface and, 444, 495–496, 583–584
fractal geometry of cyberspace, 620–622
Fractal Geometry of Nature, The, 575, 576
fractal noise. *See* noise functions; Perlin's noise function
Fractal Planetrise, 576
fractal planets. *See* MojoWorld
fractal solid textures, 447–487
clouds, 448–460
earth, 466–477
fire, 460–461, 462
iterative design of, 447–448
planetary rings, 485–487
random coloring methods, 477–485
water, 461–465
fractal sum of pulses. *See* sparse convolution noise
fractals, 429–445. *See also* MojoWorld; multifractals; procedural fractal terrains
amplification by, 436
amplitude, 433
band-limited, 434
basis function, 432–433, 450, 497–498, 582, 583
cellular texture combinations, 138–140, 141
complexity of, 431, 434–435
computer graphics and, 429
defined, 431, 571
difficulties for understanding, 430
as dilation symmetry, 431, 573–574
distribution of fractal functions, 190
fractal dimension property, 89, 432–433, 444, 495–497
fractal forgeries, 567–568
fractal increment, 432
fractional Brownian motion (fBm) and, 433
further information, 429
heuristic approach to, 431
history of fractal terrains, 575–582
lacunarity of, 433, 444, 583, 585–586
as language of visual complexity, 429, 506, 567
level of detail (LOD) in, 433–434, 437–438, 511
literature on, 579
local dimensionality in, 432
lower crossover scale, 434
mathematical history of fractal terrains, 575–576
mathematical imaging of fractal terrains, 576
misunderstanding of, 430–431
for modeling gases, 204
monofractal graphics, 445
multifractals, 438, 440–442, 569
"naturalness" of, 485
in nature, 567–568, 572–573
nonfractal complexity vs., 431
octaves in, 433–434, 444, 583, 584–585
ontogenetic modeling and, 442–444
poem about, 435
as primary building blocks, 430
procedural fBm, 436–438, 440
procedural fractal terrains, 489–506
proceduralism and, 436
random, 574–575, 582–586, 611, 613
self-similarity in, 496–497, 572–573
simplicity of, 435
software, history of, 579–580
space-filling hypertexture example, 357, 358–359
spatial frequency of basis function, 433
statistical self-similarity vs. exact self-similarity, 435
for synthesizing cloud images, 267
turbulence and, 435–436
in `turbulence` function power spectrum, 85
upper crossover scale, 434
uses for, 434–436, 506, 608–611
visual complexity and, 567–571
`fractalsum` function
power spectrum of, 86, 87
`turbulence` function vs., 86
fractional Brownian motion (fBm)
approximating, 89
basis function effects, 497–498
for color perturbation, 470
described, 599–600

Team LRN

fractional Brownian motion (*continued*)
DistNoise function based on, 450–453
fBm-valued distortion for clouds, 454–456
fBm-valued distortion for fire, 460–461, 462
fractals and, 433
homogenous fBm terrain models, 495–498
level of detail and, 437–438
monofractals using, 599–600
multiplicative-cascade multifractal variation, 440–441
in PGI, 327–329, 330
power spectrum of, 433
procedural, 436–438, 440
random fBm coloring, 477–478
as weighting function in windywave procedure, 465
fragment processing in GPUs, 287, 289
frequency of fractal basis function, 433
function fractals in MojoWorld, 602–604
functional composition
for cirrus cloud modeling, 453–454
defined, 453
in DistNoise function, 453
expression trees and, 553–554
for texture pattern generation, 26
functional flow field functions
attractors, 247–249
breeze effect using attractor, 247, 248, 252–253, 254
combinations of functions, 251–254
density_scaling parameter, 246–247
direction parameter, 246, 247
extensions of spherical attractors, 249
for flow into hole in wall, 253–254, 255
information returned by, 246–247
overview, 246–247
percent_to_use parameter, 246, 247
repulsors, 246
spherical attractor function, 248–249
spiral vortex functions, 249–251
velocity parameter, 246, 247
wind effects using, 252–253
functional flow field tables
accessing table entries, 245
combinations of functions, 251–254
functions, 246–251
overview, 245, 246
functional programming, 26
fur, volume perturbation for, 301, 302
fuzzy blobbies for modeling gases, 204
fuzzy regions
for $\texttt{bias}_b$ function for tuning, 339, 340
combining solid textures with, 338
defined, 338
$\texttt{gain}_g$ function for tuning, 339–340

GADDs (geometric atmospheric density distribution models). *See also* atmospheric models
for exponential mist, 532, 534
homogenous and isotropic, 531–532
local vs. global, 350
numerical quadrature with bounded error for radial GADDs, 542–544
overview, 530
for radially symmetric planetary atmosphere, 534–536
with Rayleigh scattering approximation, 539
trapezoidal quadrature for radial GADDs, 539–542
Gaea (Earth-like planet) model, 467–472
climactic zones, 468–469
coastline, 469
continents and oceans, 467–468
deserts, 469, 470
fractals in, 467
terran procedure, 470–472
gain function
bias function and, 39
remapping unit interval with, 38–39
$\texttt{gain}_g$ function
defined, 340
for tuning fuzzy regions, 339–340
gamma correction functions
bias, 37–38
for CRT display systems, 37, 38
gamma correction texture, 196
gases. *See also* animating solid spaces; clouds; fire; volumetric procedural modeling and texturing
animating gaseous volumes, 235–243
animating solid textured transparency, 233–235

Team LRN

basic gas shaping, 214–224
breeze effect using attractor, 247, 248, 252–253, 254
flow into hole in wall, 253–254, 255
geometry of, 211–224
hardware acceleration for smoke and fog, 297–298
helical path for, 228
historical overview of modeling approaches, 204–205
illumination model for, 207
noise and turbulence functions for, 211–214
particle systems for, 209
patchy fog modeling, 216
power and function effects on, 214–216
rolling fog animation, 238–240
self-shadowing, 207–208
sine function effects on, 215, 216
smoke column examples, 219–224, 239–243
solid spaces framework for modeling, 209–211
steam rising from teacup examples, 216–219, 220, 236–238
three-dimensional tables for controlling animation, 244–254
turbulence for simulating air currents, 238
usage in this book, 203
GDM (goal determination module), 394–395
generalized Impressionistic texture. *See* GIT (generalized Impressionistic texture) texturing system
genes, 551, 552–555
genetic algorithms, genetic programming vs., 558–559
genetic programming. *See also* genetic textures
biological analogy for, 550–552
DNA and, 551–552
examples, 557–558
expression trees, 553–556
further information, 558
future directions, 560–562
genes, 551, 552–555
genetic algorithms vs., 558–559
genomes, 551
genotypes, 551
implementation, 555
mutation, 551, 554
parameter proliferation problem and, 548
phenotypes, 551, 554
sexual reproduction, 551, 554
unnatural selection in, 552
genetic textures, 547–563. *See also* genetic programming
aesthetic *n*-space model, 548–549
basis vectors, 556
control vs. automaticity, 549–550
evolutionary biological model, 550–555
expression trees for, 553–556
future directions, 560–562
genetic programming and genetic art examples, 557–558
genetic programming vs. genetic algorithms, 558–559
library of genetic bases, 556–557
parameter proliferation problem, 547–548
root node interpretation, 555–556
genomes, 551
genotypes, 551
geometric atmospheric density distribution models. *See* GADDs (geometric atmospheric density distribution models)
geometric modeling
advanced techniques, 2–3
bump map geometry, 42
evolution of, 2–3
star geometry, 47
geometric normal, GPU support lacking for, 106
geometrical calculations, 7
geometry mapping with PGI, 326–327
geometry of gases, 211–224
geometry, procedural synthesis of. *See* procedural synthesis of geometry
gesturing. *See* textural limb animation
GIT (generalized Impressionistic texture) texturing system, 478–480
Impressionistic image processing filter, 479–480
mathematical model underlying, 478–479
overview, 478–479
global shading models, 8
`gnoise` function. *See* gradient noise
goal determination module (GDM), 394–395
GPUs. *See also* hardware acceleration of procedural techniques
assembly languages for, 100–101
blending capability, 290
commercially supported programming languages, 111

Team LRN

basic gas shaping, 214–224
breeze effect using attractor, 247, 248, 252–253, 254
flow into hole in wall, 253–254, 255
geometry of, 211–224
hardware acceleration for smoke and fog, 297–298
helical path for, 228
historical overview of modeling approaches, 204–205
illumination model for, 207
noise and turbulence functions for, 211–214
particle systems for, 209
patchy fog modeling, 216
power and function effects on, 214–216
rolling fog animation, 238–240
self-shadowing, 207–208
sine function effects on, 215, 216
smoke column examples, 219–224, 239–243
solid spaces framework for modeling, 209–211
steam rising from teacup examples, 216–219, 220, 236–238
three-dimensional tables for controlling animation, 244–254
turbulence for simulating air currents, 238
usage in this book, 203
GDM (goal determination module), 394–395
generalized Impressionistic texture. *See* GIT (generalized Impressionistic texture) texturing system
genes, 551, 552–555
genetic algorithms, genetic programming vs., 558–559
genetic programming. *See also* genetic textures
biological analogy for, 550–552
DNA and, 551–552
examples, 557–558
expression trees, 553–556
further information, 558
future directions, 560–562
genes, 551, 552–555
genetic algorithms vs., 558–559
genomes, 551
genotypes, 551
implementation, 555
mutation, 551, 554
parameter proliferation problem and, 548
phenotypes, 551, 554
sexual reproduction, 551, 554
unnatural selection in, 552
genetic textures, 547–563. *See also* genetic programming
aesthetic *n*-space model, 548–549
basis vectors, 556
control vs. automaticity, 549–550
evolutionary biological model, 550–555
expression trees for, 553–556
future directions, 560–562
genetic programming and genetic art examples, 557–558
genetic programming vs. genetic algorithms, 558–559
library of genetic bases, 556–557
parameter proliferation problem, 547–548
root node interpretation, 555–556
genomes, 551
genotypes, 551
geometric atmospheric density distribution models. *See* GADDs (geometric atmospheric density distribution models)
geometric modeling
advanced techniques, 2–3
bump map geometry, 42
evolution of, 2–3
star geometry, 47
geometric normal, GPU support lacking for, 106
geometrical calculations, 7
geometry mapping with PGI, 326–327
geometry of gases, 211–224
geometry, procedural synthesis of. *See* procedural synthesis of geometry
gesturing. *See* textural limb animation
GIT (generalized Impressionistic texture) texturing system, 478–480
Impressionistic image processing filter, 479–480
mathematical model underlying, 478–479
overview, 478–479
global shading models, 8
`gnoise` function. *See* gradient noise
goal determination module (GDM), 394–395
GPUs. *See also* hardware acceleration of procedural techniques
assembly languages for, 100–101
blending capability, 290
commercially supported programming languages, 111

Team LRN

GPUs (*continued*)
computational precision issues, 288
CPUs vs., 100, 102, 109
data types, 108–109
dot product capability, 290
filtering capability, 290
flexibility issues, 288–289
fragment processing in, 287, 289
future hardware and programming languages, 130–131, 410
interpolation capability, 290
language interface, 288
levels of operation in the pipeline, 289
limitations and restrictions, 102–103, 106–110
memory bandwidth and performance tuning, 110
migrating procedural techniques from CPUs to, 287–289
parallelism in, 102, 103, 106–108
real-time procedural solid texturing on, 421–425
real-time procedural techniques and, 1
resource limits, 109
REYES shading contrasted with, 103–106
separation of surface and light properties and, 117
SIMD computation model in, 107–108
SPMD computation model in, 108
storage issues, 103, 289
vertex processing in, 287, 289
virtualization support lacking in, 109

gradient noise
2D slice of, 74
flow noise, 384–387
generating value for single integer lattice point, 76
graph of, 73
initializing table of pseudorandom gradient vectors, 73–75
lattice noise, 69–70
lookup table for, 181
overview, 72–77
Perlin's function, 69, 73, 75–76, 340–348
for perturbed regular patterns, 89–90
power spectrum of, 75, 77
trilinear interpolation, 76–77
value-gradient noise, 77–78
Graphics Gems III, 180
graphics processors. *See* GPUs; hardware acceleration of procedural techniques
grass modeling
bounding volumes for, 331
geometry mapping for, 326–327
iterative instancing for, 322–323
meadows, 327–329
PGI for, 322–323, 326–329, 331
grid tracing, height fields and, 494
GUI for textures. *See* user interface for textures

hardware acceleration of procedural techniques, 287–302. *See also* GPUs
animated real-time effects, 291
common acceleration techniques, 289–291
dummy geometry and depth culling, 290
example accelerated/real-time textures and models, 291–301
general issues, 287–289
marble texture, 292–297
migrating procedural techniques from CPUs to GPUs, 287–289
multilevel procedural models, 290–291
noise functions, 291
precomputation for, 289–290
real-time clouds and procedural detail, 298–301, 302
real-time procedural solid texturing, 416, 421–425
smoke and fog, 297–298
tasks beyond designers' intentions, 290
texture mapping, 208, 209
turbulence functions, 291–292
hardware texture mapping
for modeling and rendering gases, 209
for shadowing gases, 208
hashing in lattice noise generation, 69–70
Heidrich/Banks anisotropic shader
example using explicit computations, 113–115, 116
example using implicit computations, 115–116
explicit vs. implicit version, 116
overview, 112–113
height fields
defined, 491
as displacement maps, 511
file formats, 494–495

Team LRN

for light diffusion effects simulation, 204
post spacing side effects, 509
in QAEB tracing, 511, 516–517
ray tracing and, 494
single altitude value per grid point in, 493
speedup scheme for, 513, 516–517
as storage scheme for terrains, 491
helical paths
for animating gaseous volumes, 228, 236–243
for fog animation as solid textured transparency, 233
for marble animation, 232–233
for rolling fog animation, 238–239
for smoke rising animation, 239–243
for smoke simulation, still image, 220, 222–224
for space-filling fog, 298
for steam rising from teacup animation, 236–238
Henyey-Greenstein functions for illumination of gaseous phenomena, 207
Hermite blending functions for bump mapping ridge, 186
Hermite noise
2D slice of, 74
described, 78
graph of, 73
power spectrum of, 75, 78
Hermite spline interpolation in noise functions, 490, 498
heterofractal function in MojoWorld, 601, 602
heterogeneous terrain models, 498–506
hybrid multifractal terrain, 502–505
multiplicative multifractal terrains, 505–506
real landscapes and, 498, 500
smooth valleys at all altitudes, 502–505
smoother low-lying areas, 500–502
statistics by altitude, 500–502
`Hetero_Terrain` procedure, 500–501
hierarchy of surflets, 381–382
high frequencies. *See also* aliasing; antialiasing; Nyquist frequency
aliasing and, 53, 54, 370
edge events as sources of, 370
`if` function generation of, 47
non-edge high-frequency events, 370
`step` function generation of, 54, 57
high-albedo reflection models
for clouds, 266
low-albedo models vs., 205
high-contrast filter for antialiasing, 371–372, 373–374
HighEnd3D Web site, 284
homogenous, defined, 438
homogenous fBm terrain models, 495–498
basis function effects, 497–498
fractal dimension, 495–497
homogenous procedural fBm, 438
HSV color, transforming RGB to, texture for, 196
Human Genome Project, 551
humanlike figures
textural limb animation for emotive gesturing, 387, 391–395
texture for facial movement, 395–409
hurricane, ontogenetic modeling of, 456–458
hybrid multifractal terrains, 502–505
`HybridMultifractal` procedure, 502–504
`RidgedMultifractal` procedure, 504–505
hyperspace in MojoWorld, 588–590
complexity and, 588–589
defined, 588
dimension defined, 588
Parametric Hyperspace, 589–590
hypertextures
animating, 254–256
architexture, 359–362, 363
defined, 338
editing levels for, 352–353
egg shape for, 354
explosions example, 354–356
interaction with, 352–353
life forms example, 355, 356, 357
liquid flowing through hole in wall, 254–255
methods of combining fuzzy shapes with solid textures, 338
NYU torch example, 362, 364
QAEB-traced, 520–525, 526
raymarcher renderer for, 348–352
smoke examples, 364–368
solid textures as precursor to, 337–338
space-filling fractals example, 357, 358–359
sphere shape for, 353–354
surflets for storing, 376–384
volumetric marble formation, 255–256

hypertextures (*continued*)
woven cloth example, 357, 359, 360, 361
z-slicing, 35

`if` function (RenderMan)
antialiasing and, 28
high frequencies generated by, 47
replacing with `step` function, 28–29
ifpos pseudofunction, 371, 372–373
illumination models. *See* shading models
image maps
converting 3D textures to, 197–199
textures vs., 196–197
"Image Synthesizer, An," 590
image textures
perturbed, 90–91
PhotoRealistic RenderMan antialiasing for, 56
procedural textures vs., 14–15
implicit functions for cloud modeling, 269, 270–272
implicit models, 10
implicit primitive animation for clouds, 280–283
implicit procedures
defined, 12–13
explicit procedures vs., 12–14
in RenderMan shading language, 15
implicit surfaces, 3
imposters
for cloud modeling, 268–269
defined, 268
for gases, 209
index aliasing
defined, 159
modeling input distribution, 161–162
planetary rings example, 163–166
reducing, 160–163
sources of, 158–159
sum table for, 162–163
transformation of scalar functions and, 159–160
inductive instancing, 322–323
`init_density_function` for raymarcher renderer, 350
instances
defined, 315
inductive instancing, 322–323
parameter passing and, 321–322
of scene graphs, 315
integrals, antialiasing using, 63–64, 65
integration schemes
numerical quadrature with bounded error for radial GADDs, 542–544
as requirement for atmospheric models, 529, 530
trapezoidal quadrature for radial GADDs, 539–542
intelligent textures, 200
interactivity
cloud modeling approaches for, 267–268
for hypertextures, 352–353
in previewer for textures, 194
real-time vs. offline shading and, 98, 99, 100
intermediate representation, 313
interpolation capability of GPUs, 290
inverse Fourier transform
defined, 49
domain change from, 49
noise generation using, 82–83
for spectral synthesis, 49–51
irregular procedural textures, 67–94
noise functions for, 67–83
perturbed image textures, 90–91
perturbed regular patterns, 89–90
pseudorandom number generation and, 67–68
random placement patterns, 91–94
RenderMan noise functions for, 83–85
spectral synthesis, 85–89
time-dependent textures, 84
white noise and, 67–68
isocurves, 13
isosurfaces
defined, 3
implicit models, 10
overview, 13
surflets and, 383
isotropic property
of cellular texturing function, 142, 143, 145–147
of `noise` function, 68, 180
of procedural fBm, 438
iterated function systems, 320
iteration
in fractal solid texture design, 447–448
iterative instancing, 322–323
in scientific discovery, 447–448
in shader development process, 99, 130

jaggies. *See* aliasing; antialiasing
Java, procedural geometric modeling and, 333

Team LRN

jet stream warping for clouds, 279–280

Kelvin-Helmholtz shearing in clouds, 266
KISS (Keep it simple, stupid) principle, 443
knot values in `spline` function (RenderMan), 34
Koch snowflake. *See* von Koch snowflake

labels in L-system, 312
lacunarity
  defined, 89, 583
  of fractals, 433, 444, 583, 585–586
  noise artifacts and, 180
Lambertian (diffuse) model, 708
landscapes. *See* atmospheric models; earth textures; MojoWorld; plant modeling; procedural fractal terrains
language interface, CPUs vs. GPUs, 288
lapped textures, 11
lattice convolution noise
  2D slice of, 74
  graph of, 73
  implementation of, 78–80
  power spectrum of, 75, 80
lattice noises, 69–80
  artifacts, 180
  frequency multiplier and lattice artifacts, 88–89
  gradient noise, 72–77
  hashing technique, 69–70
  integer lattice, 69
  lattice convolution noise, 78–80
  for modeling gases, 211–214
  optimizing, 214
  overview, 69–70
  PRNs for, 69
  value noise, 70–72
  value-gradient noise, 77–78
laughing, 407–408. *See also* texture for facial movement
layering
  RGBA textures and, 26
  for texture pattern generation, 25–26
lazy evaluation algorithms for procedural geometry
  client-server relationship and, 314
  described, 305, 314
  L-system and, 315
  for spatial coherence data structures, 314–315
LED display, texture for, 196–197
length-weighted atlas, 418
lerping
  layering texture patterns and, 26
  `LERP` function in antialiasing, 371, 373
  in rasterization phase for real-time procedural solid texturing, 414
level of detail. *See* LOD (level of detail)
life forms hypertexture example, 355, 356, 357
light properties, 116–117
light shaders, surface shaders vs., 117
lighting
  photographic texture images and, 22–23
  surflets for, 376–377
lighting models. *See* shading models
limb animation. *See* textural limb animation
linear basis function (MojoWorld), 596
linear interpolation. *See* lerping
liquid. *See also* water
  flow into hole in wall, 253–254, 255
  flow noise for, 384–387, 388–389
  volumetric marble formation, 255–256
LISP, functional composition in, 26
Listerine (RenderMan examples), 18–19
local dimensionality in fractals, 432
local shading models, 8
LOD (level of detail)
  in fractals, 433–434, 437–438, 511
  procedural fBm and, 437–438
low-albedo reflection models for clouds, 266
  high-albedo models vs., 205
lower crossover scale
  of fractals, 434
  in MojoWorld, 607
low-pass filtering
  analytic prefiltering techniques, 56–57, 61–66
  antialiasing and, 54
  clamping, 56, 59–61
  defined, 54
  in lattice noise generation, 69
  sampling rate determination, 57–59
  of white noise, 68
L-system, 307–312. *See also* procedural geometric instancing (PGI); scene graphs
  axioms, 308
  bicubic patch modeling, 311
  bush modeling, 318–320
  color and texture specification, 311–312
  context-free and context-sensitive aspects, 308

Team LRN

L-system (*continued*)
data amplification paradigm in, 312–314
described, 305
development of, 307
intermediate representation in, 313
labels for debugging or annotation, 312
lazy evaluation and, 315
operating system command execution, 312
for particle systems, 313–314
PGI compared to, 329–330
PGI enhancements, 321–330
for plant modeling, 308–310, 311, 313
polygon modeling, 311
productions, 307, 308
scene graphs, 315–321
shortcomings of, 305–306
turtle graphics symbols in, 305, 306, 308–312
von Koch snowflake modeling, 308, 310–311
luminosity, volume rendering with surface textures and, 196
`luna` procedure, 474–476

Mandelbrot set, texture using, 197
"Manhattan" distance metric in cellular texturing, 148
mapping from unit interval to itself, 37–39
`marble` function, 229, 292
marble texture
animated volumetric marble formation, 255–256
animating by changing solid space over time, 229–232
animating by moving through solid space, 232–233
formation from banded rock, 229–232
frequency multiplier for, 88–89
hardware acceleration for, 292–297
NVIDIA Cg language implementation, 292–296
for solid texture animation, 229–232
spectral synthesis for, 87–89
`marble_color` function, 88, 229
`marble_forming` procedure, 230–231
`marble_forming2` procedure, 231–232
`Marble_Vertex.cg` shader, 294–296
master, defined, 315
matte shading model, `plastic` shader and, 21
`max` function (RenderMan), 28–29
Maya animation system
cloud modeling in, 267, 284, 285
MEL scripts, 284, 285
volumeGas plug-in, 284, 285
Web sites, 284
meadows, PGI for modeling, 327–329
memory
GPU resource limits, 109
GPU restrictions on access, 107
performance tuning and bandwidth, 110
meta-balls, 3
MetaCreations Skunk Works, 493
metal shading model, `plastic` shader and, 21–22
"Methods for Realistic Landscape Imaging," 576
Mie scattering, aerial perspective and, 530
`min` function (RenderMan), 28–29
MIP mapping, atlases supporting, 418–419
mist, exponential, 532, 534
`mix` function (RenderMan), 25
`mod` function (RenderMan), 31–33
MojoWorld, 565–615
aliasing from function fractals, 604
basis functions, 450, 582, 583, 590–597
building a virtual universe, 571
checkerboard basis function, 596–597
for clouds, 611
crossover scales, 606–607
dimensions of domain and range functions, 587–588
displacement maps, 610–611
distorted fractal functions, 606
domain distortion, 604–606
driving function parameters with functions, 607
ease of, 447
experimenting with, 613–614
fractal dimension and, 582–584
fractals and visual complexity, 567–571
fractals overview, 571–575
function fractals, 602–604
future work, 581–582, 614–615
Graph Editor, 606, 607
heterofractal function, 601, 602
history of fractal terrains, 575–582
hyperspace, 588–590
lacunarity of fractals, 583, 585–586

linear basis function, 596
MojoWorld Generator, 590
MojoWorld Texture Editor, 589, 590, 604
monofractals, 599–600
mountain building, 567–568
multifractals, 600–602, 603
for nebulae, 611, 612
octaves of fractals and, 583, 584–585
Perlin basis function, 590–591, 592, 593, 594
photorealistic renderer, 585, 617–618
planet building, 568–570, 611, 612
Pro UI, 566–567, 606
random fractals, 582–586
real-time renderer, 585, 617
reason for inclusion in this book, 565–566
ridged basis function, 590–591, 593, 594
seed tables, 597–599
SIGGRAPH 2001 presentation of, 200
sine wave basis function, 596
sparse convolution basis function, 593, 595
steps basis function, 596, 597
surface textures, 589, 608–609
for terrains, 609–610
texture engine, 589
transporter coordinates, 589
Voronoi basis function, 591, 593, 594, 597, 599
MojoWorlds, 467
`molten_marble` procedure, 255–256
monofractals in MojoWorld, 599–600
Monte Carlo sampling method
in Perlin's noise function, 342
in volume-rendering algorithm, 206
Moon texture. *See* Selene (Moon) texture
motion blur, object space vs. screen space shading and, 105–106
mouse abdomen visualization, 122
`moving_marble` procedure, 232–233, 236
"multicolor" texture, 482–485
goal of, 482–483
`multicolor` shader code, 484
"naturalness" of, 485
steps for, 483–484
multifractals, 438, 440–442. *See also* MojoWorld; procedural fractal terrains
defined, 440
described, 569
fBm variation, 440–441
hybrid multifractal terrain, 502–505
MojoWorld, 600–602, 603
multiplicative cascades and, 440
multiplicative multifractal terrains, 505–506
procedural textures and, 442
realism and, 438, 440
terrain patch example, 441
multipass pixel shader implementation of noise function, 422–425
multiplicative cascades, additive cascades vs., 440
mutation, 551, 554

Natural Graphics' Natural Scene Designer, 580
Navier-Stokes solutions for modeling gases, 204
nearest-neighbor filter, seam artifacts and, 420
nebulae, MojoWorld for modeling, 611, 612
`Ng` variable, RenderMan vs. real-time hardware rendering and, 106
`noise` function (RenderMan), 83–84
noise functions, 67–83. *See also* cellular texturing; Perlin's noise function
for 2D cloud textures, 267–268
2D slices of, 74
4D functions, 84, 181
artifacts from, 180–181
cellular texturing compared to, 135–136
in cloud modeling, 270, 298–299
color splines and, 189
distorted (`DistNoise`), 450–453
enhancements and modifications, 179–182
explicit noise algorithms, 82–83
flame shader based on, 124–129
flow noise, 384–387
Fourier spectral synthesis, 82–83
for gases, 211–214
generating, 67–68
gradient noise, 72–77
graphs of, 73
hardware acceleration for, 291
Hermite spline interpolation in, 490, 498
lacunarity, 89, 180
lattice convolution noise, 78–80
lattice noises, 69–70
in layering facial movement, 405–408

noise functions (*continued*)
lookup table for, 181
for marble texture, 87–89
multipass pixel shader implementation, 422–425
need for, 67
noise value range and distribution for, 85
normalizing noise levels, 189–190
performance issues, 194–195
for perturbed image textures, 90–91
for perturbed regular patterns, 89–90
with PGI, 327
as pink noise, 68
power spectra of, 75
properties of ideal function, 68
random placement patterns using, 91–94
Rayshade implementation, 422–425
in real-time procedural solid texturing, 421–425
RenderMan functions, 83–85
rotation matrices for hiding artifacts, 180, 181
for smoke and fog, 297, 298
sparse convolution noise, 80–82
spectral synthesis with, 85–89
as surflet, 379
value noise, 70–72
value-gradient noise, 77–78
vector-valued (`VecNoise`), 450–451
versatility of, 135
for volumetric cloud modeling, 270
Ward's Hermite noise function, 73, 74, 75
white noise and, 67–68
noise textures, development of, 11
nonprocedural textures, procedural vs., 12
normal vectors, transforming between texture spaces, 46
normalizing noise levels, 189–190
n-space model, 548–549, 559
`ntransform` function (RenderMan), `transform` function vs., 46
NVIDIA's Cg language, 111, 292–296
Nyquist frequency
aliasing and, 53–54
clamping and, 60
defined, 53
MojoWorld function fractals and, 602–603
PhotoRealistic RenderMan antialiasing scheme and, 56
QAEB tracing and, 511, 512
white noise and, 68
NYU torch hypertexture example, 362, 364

object space
described, 24
screen space vs., for shading, 103–106
transformation to world space with PGI, 323–324
Occam's Razor, 443, 530
octaves in fractals, 433–434, 444, 583, 584–585
offline programmable shading, real-time programmable shading vs., 98–100
ontogenetic modeling
defined, 442
fractals and, 442–444
of hurricane, 456–458
Occam's Razor and, 443
realism and, 444
semblance in, 443
teleological modeling vs., 442
opacity
`plastic` shader output, 21
solid textured transparency, 233–235
in volume-rendering algorithm for gases, 206
OpenGL
Direct3D vs., 102
lighting model, 118
pixel texture extension, 422
for real-time cloud rendering, 267
OpenGL Programming Guide, 102
OpenGL Shader project, 131
operands in GPUs, precision of, 288
optical depth, 531
optical paths, 531, 534
optimizing. *See also* performance
hypertexture smoke, 367–368
lattice noises, 214
spot size, 173–175
outscattering
defined, 531
Rayleigh, 536
oversampling. *See* supersampling

parallelism
CPUs vs. GPUs and, 102
GPU restrictions due to, 103, 106–108
parameter passing in PGI, 321–322

Team LRN

parameter proliferation problem, 547–548
parametric control, 2, 14
Parametric Hyperspace, 589–590
parametric patches, 8
particle systems
  animation using, 257–261
  attributes of particles, 257
  for cloud modeling, 269, 282–283
  initial shape, 257–258
  L-system for, 313–314
  movement of particles, 258
  overview, 3
  particle creation procedure, 257
  probabilistic rendering, 259, 260
  processes for each time step, 257
  rendering problems, 259, 261
  structured, 257, 258–259, 260
  uses for, 209, 257, 259
  for wall of fire, 257, 258
pattern generation
  advantages of procedural generation, 23
  brick texture example, 39–41
  bump-mapped brick example, 41–46
  checkerboard pattern, 39
  defined, 20
  functional composition technique, 26
  hiding regularity and periodicity, 51
  irregular patterns, 83–94
  layering technique, 25–26
  perturbed image textures, 90–91
  perturbed regular patterns, 89–90
  photographic texture images for, 22–23
  primitive functions as building blocks, 27–51
  random placement patterns, 91–94
  spectral synthesis technique, 48–51, 85–89
  star texture example, 46–48, 49
  writing procedural generators, 23–24
pelting, described, 10–11
penumbra, self-shadowing with, 382–383
percentage closer filtering, described, 9
performance. *See also* hardware acceleration of procedural techniques; optimizing
  antialiasing efficiency issues, 157
  assembly language vs. high-level language for shading and, 101
  in cellular texturing, speed vs. isotropy, 146–147
  as critical for real-time shading, 98, 99–100
  early vs. late binding and, 117–118
  hardware improvements and, 97
  height field speedup scheme, 513, 516–517
  memory bandwidth and performance tuning, 110
  procedural texture efficiency issues, 194–195
  of QAEB tracing, 510, 511, 518, 520
  of ray-traced self-shadowing, 207–208
  real-time procedural solid texturing issues, 425–426
  target hardware and, 130
  texture size and, 110
periodic functions (RenderMan)
  `ceil`, 33
  `cos`, 31, 32
  `floor`, 32–33
  lacunarity and, 180
  making other functions periodic, 32
  `mod`, 31–32
  `sin`, 31, 32
Perlin basis function (MojoWorld), 590–591, 592, 593, 594
Perlin's noise function
  construction of, 340–347
  `DistNoise` function based on, 450
  finding current cubical "cel" location, 341
  finding the pseudorandom wavelet at each cel vertex, 341–342
  folding function, 343
  as fractal basis function, 432–433
  gradient distribution, 348
  gradient table, 342–343
  interpolation polynomial improvement, 347
  in MojoWorld basis function, 590–591, 592, 594
  nonbiased index of *G*, 343–344
  overview, 340–341
  performance improvements, 348
  point evaluation in, 489–490
  procedural fBm and, 437
  pseudorandom permutation table, 343
  in QAEB-traced hypertextures, 521
  in real-time procedural solid texturing, 421–423

Team LRN

Perlin's noise function (*continued*)
reducing grid-oriented artifacts, 348
as seminal function, 69, 73
steps in computation, 341
uniformly distributed unit gradients generated by, 75–76
wavelet coefficients, 342–344
wavelet evaluation, 344–347
wavelet properties, 342
`perm` array in lattice noise generation, 69–70
perturbance
perturbed image textures, 90–91
perturbed regular patterns, 89–90
texture for, 197
volume perturbation for cloud modeling, 299–301
PGI. *See* procedural geometric instancing (PGI)
phenotypes, 551, 554
photographic texture images, 22–23
PhotoRealistic RenderMan
antialiasing in, 55–56
REYES algorithm, 103
sampling interval changes in, 59
screen space use by, 103
pink noise, 68
pipelines
computational precision issues, 288
levels of operation in, 289
REYES (PRMan) vs. real-time graphics hardware, 103–104
PixelFlow project, 131
planetary atmosphere. *See* atmospheric models
planetary rings, fractal solid textures for, 485–487
planetary rings antialiasing example, 163–166
`planetclouds` procedure, 453–455
plant modeling. *See also* MojoWorld
bounding volumes for, 331–332
bushes, 318–320
crop circles, 326
grass, 322–323, 326–329, 331
L-system for, 308–310, 311, 313, 318–320
meadows, 327–329
PGI for, 322–323, 324–329, 331–332
trees, 309–310, 324, 325, 331–332
tropism, 324–326
`plastic` shader, 20–22
plateau width parameter for bump mapping ridge, 184
`pnoise` function (RenderMan), 84–85
point evaluation
context-independence of, 490
Fourier synthesis vs., 490
in Perlin noise-based procedural fractal construction, 489–490
polygon subdivision vs., 490
rounded terrain capabilities of, 490–491
Poisson distribution in cellular texturing, 145–147
polygon subdivision
basis function for, 497
for mountain modeling, 489
point evaluation vs., 490
terrain creases as artifacts, 498
polygons
cloud modeling using, 268
L-system modeling of, 311
object space vs. screen space shading and, 104–105
post spacing of height fields, 509
power functions
gas shape and, 215–216
for tuning procedural textures, 339
power spectra
of fractional Brownian motion (fBm), 433
of noise functions, 75
precomputation for procedural techniques, 289–290
previewer for textures, 194
Primitive Itch's ShaderLab2 package, 130
PRMan. *See* PhotoRealistic RenderMan
PRNs (pseudorandom numbers)
in cellular texturing, 143
for lattice noises, 69
table for noise functions, 69
for value noise, 70–71
for white noise, 67–68
Pro UI of MojoWorld, 566–567, 606
procedural, defined, 12
procedural city technique, 200
procedural cloud animation, 279–280
procedural evaluation phase for real-time procedural solid texturing, 414, 425
procedural fBm, 436–438, 440
procedural fractal terrains, 489–506. *See also* earth textures; MojoWorld
advantages of point evaluation, 489–491
basis function effects, 497–499
fractal dimension, 495–497
height fields, 491–495

Team LRN

heterogeneous terrain models, 498–506
homogenous fBm terrain models, 495–498
hybrid multifractal terrain, 502–505
multiplicative multifractal terrains, 505–506
rounded terrain capabilities, 490–491, 498
statistics by altitude, 500–502
procedural geometric instancing (PGI), 321–330
accessing world coordinates, 323–324
bounding volumes, 330–332
crop circles example, 326
described, 321
fractional Brownian motion, 327–329
front-to-back ordering, 330
geometry mapping example, 326–327
inductive instancing example, 322–323
levels of detail, 329
L-systems compared to, 329–330
meadows example, 327–329
noise function with, 327
parameter passing, 321–322
random number function, 327
trees example, 331–332
tropism example, 324–326
procedural modeling of gases. *See* volumetric procedural modeling and texturing
procedural shape synthesis, 387, 390
procedural synthesis of geometry, 305–334
applications, 305
bounding volumes, 330–332
data amplification algorithms, 305, 312–314
flow of data, 312
future work, 333–334
lazy evaluation algorithms, 305, 314–315
L-system, 305–306, 307–312
overview, 332–333
paradigms governing, 312–315
procedural geometric instancing (PGI), 321–330
scene graphs, 307, 315–321
Web and, 333
procedural techniques. *See also specific techniques*
defined, 1
implicit vs. explicit, 12–14
migrating from CPUs to GPUs, 287–289
overview, 1–2
precomputation for, 289–290
procedural textures (overview). *See also* texture design methods; volumetric procedural modeling and texturing
advantages of, 14
aliasing in, 55–56
antialiasing methods, 56–67
antialised rendering of, 369–376
brick texture example, 39–41
bump-mapped brick example, 41–46
checkerboard pattern, 39
defining characteristics, 12
disadvantages of, 14–15
efficiency issues, 194–195
explicit vs. implicit methods, 12–14
functional composition technique, 26
hiding regularity and periodicity, 51
historical overview, 11–12
image texture vs., 14–15
irregular, 67–94
isocurve or isosurface method, 13
layering technique, 25–26
multifractal models, 442
nonprocedural vs., 12
pattern generation, 20, 22–24, 25–26
primitive functions as building blocks, 27–51
renderer antialiasing schemes and, 55
shading models, 20–22
spectral synthesis technique, 48–51
star texture example, 46–48, 49
texture spaces, 24–25
volume rendering with surface textures, 195–196
productions in L-system, 307, 308
pseudoadvection in flow noise function, 386–387
pseudorandom numbers. *See* PRNs (pseudorandom numbers)
psychedelic clouds, 525, 526
`puffyclouds` procedure, 448–449
pyroclastic flow, 523–524

QAEB rendering for procedural models, 509–526
antialiasing, 516
C code implementation, 625–626

Team LRN

QAEB rendering for procedural models (*continued*)
clouds, 520–525, 526
error in the algorithm, 513–514
fireballs, 525
implicit models and, 510
intersection point calculation, 515
meaning and pronunciation of acronym, 510, 511
near and far clipping planes, 514–515
Nyquist limit and, 511, 512
performance, 510, 511, 518, 520
prior art, 512–513
problem statement, 511–512
pyroclastic flow, 523–524
QAEB algorithm, 513–514
QAEB tracing overview, 510–511
QAEB-traced hypertextures, 520–523
reflection and refraction, 517–518
shadows, 517
speedup scheme for height fields, 513, 516–517
stride length, 521–522
surface normal construction, 516
quad tree spatial subdivision, height fields and, 494
Quake III game engine, 131
quasi-analytic error-bounded ray tracing. *See* QAEB rendering for procedural models

radar texture, 197
radial coordinate distance metric in cellular texturing, 148
radiosity as global shading model, 8
random coloring methods, 477–485
fBm coloring, 477–478
GIT texturing system, 478–480
"multicolor" texture, 482–485
random fractals
basis function, 582, 583
constructing, 582–586
expressive vocabulary of, 611, 613
fractal dimension, 582–584
lacunarity of, 585–586, 593
octaves, 583, 584–585
overview, 574–575
random placement patterns
defined, 91
`noise` function for, 91–94
one-star-per-cell version, 92–93
storing bomb positions in table, 91
version improving clipping and randomness, 93–94
randomness. *See also* noise functions; PRNs (pseudorandom numbers)
deterministic nature of computers and, 574
MojoWorld seed tables for, 597–599
PGI functions, 327
in setting texture parameters, 193
true vs. pseudorandomness, 67, 574–575
range function dimensions in MojoWorld, 587
rasterization phase for real-time procedural solid texturing, 414, 416, 419, 425–426
ray tracing. *See also* QAEB rendering for procedural models
antialiasing by stochastic ray tracer, 55
as global shading model, 8
height fields and, 494
implicit geometric models in, 13
implicit vs. explicit procedures in, 13
raymarcher renderer, 348–352
ray-surface intersection point calculation, 515
reflection mapping as, 9
scanline rendering vs., 511
for self-shadowing of gases, 207–208
in surflets, 379–380, 382–383
for volume rendering of gases, 205–206
Rayleigh scattering
aerial perspective and, 530
color perspective due to, 529
in *Fractal Mandala,* 533
minimal approximation, 536, 539
raymarcher renderer, 348–352
overview, 348–349
system code, 349–350
user-defined functions, 350–352
Rayshade implementation of noise function, 422–425
reaction-diffusion textures, intelligent, 200
realism
complexity and, 434–435
heterogeneous terrains for, 498, 500
multifractal functions and, 438, 440
"naturalness" of fractals, 485

Team LRN

ontogenetic modeling and, 444
real-time clouds, 298–301
real-time procedural solid texturing, 413–427
algorithm, 413–416
applications, 425–426
area-weighted mesh atlas, 417–418
atlas based on clusters of proximate triangles, 418, 419
atlas construction for, 416–419, 425
bilinear filtering, 420–421
hardware acceleration, 416, 421–425
implementing, 421–425
length-weighted atlas, 418
multipass pixel shader implementation of noise function, 422–425
nearest-neighbor filter, 420
noise functions in, 421–425
parameterization, 413
performance issues, 425–426
procedural evaluation phase, 414, 425
rasterization phase, 414, 416, 419, 425–426
scaling component of distortion, 416
seam artifacts, avoiding, 419–421
solid texture coordinates, 413–414
spatial coordinates, 413
texture filtering, 419–421
texture mapping phase, 415–416
uniform meshed atlas, 416–417
view independence, 425
real-time programmable shading, 97–132
flame shader, noise-based procedural, 124–129
future GPU hardware and programming languages, 130–131
GPU architecture, 102–103
hardware data types, 108–109
Heidrich/Banks anisotropic shader, 112–116
high-level shading language advantages, 100–101
interactivity and, 98, 99, 100
interface between shaders and applications, 118–121
iterative nature of development process, 99, 130
knowledge required for the reader, 101–102
literature review, 131
memory bandwidth and performance tuning, 110
object space shading vs. screen space shading, 103–106
offline programmable shading vs., 98–100
parallelism, 106–108
performance as critical issue for, 98, 99
resource limits, 109
simple examples, 111–116
Stanford shading language example, 111–112
strategies for developing shaders, 129–130
surface and light shaders, 116–118
volume-rendering shader, 121–124
rectangular pulse, `step` function for, 28, 29
recurrent iterated function systems, 320
reflection, QAEB tracing and, 517–518
reflection mapping, 9
reflection models. *See* shading models
refraction, QAEB tracing and, 517–518
renderers
antialiasing schemes and procedural textures, 55
flow of data and, 312
image display while rendering, 194
raymarcher, 348–352
spot size calculation by, 166–167
rendering
antialised rendering of procedural textures, 369–376
efficiency issues, 194–195
image display during, 194
particle systems, 259, 261
previews, 194
texture as 2D image, 176
volume, 121–124, 195–196
volumetric rendering system, 205–208
RenderMan
development of, 1
PhotoRealistic, 55–56, 59, 103
RenderMan Companion, The, 102
RenderMan shading language. *See also specific functions*
brick texture example, 39–41
bump-mapped brick texture example, 43–46
cloud simulation using spectral synthesis, 50–51
conditional functions, 27–31
`du` and `dv` variables, 58
ease of, 447
examples using, 16–19

Team LRN

RenderMan shading language (*continued*)
further information, 15, 102
multiplying colors in, 26
noise functions, 83–85
OpenGL lighting model and, 118
overview, 15–16, 19
periodic functions, 31–33
`plastic` shader in, 20–22
real-time shading and, 101–102
star texture example, 47–48
surface and light properties separated in, 117
texture spaces, 24–25
translating into C code, 19
reptile hide texture, 139
repulsors, 246
resolution
image textures vs. procedural textures and, 14
low-resolution previews, 194
surflet hierarchy, 381–382
resource limits in GPUs, 109
REYES algorithm (PRMan), 103–106
RGB color. *See also* color
multiplying colors together, 26
random fBm coloring, 477–478
in RenderMan shading language, 118
transforming to HSV using a texture, 196
RGBA color
layering texture patterns and, 26
in OpenGL lighting model, 118
in Stanford shading language, 118
ridge for bump mapping, 183–187
algorithm, 187
bevel shape parameters, 184–186
Hermite blending functions, 186
non-geometric applications, 187
plateau width parameter, 184
ridge shapes for different slope controls, 185
ridge width parameter, 183–184
simple ridge, 183, 184
ridged basis function (MojoWorld), 590–591, 593, 594
`RidgedMultifractal` procedure, 504–505
`Rings` procedure, 485, 487
`ripples` procedure, 462–463
`rising_smoke_stream` procedure, 239–243
rotation along helical path, 228
rotation matrices, noise artifacts and, 180, 181
roughness. *See* fractal dimension
RTSL. *See* Stanford shading language

sample point, shading, 15
sampling
defined, 52
Monte Carlo method, 206
stochastic, 55–56, 67
supersampling, 54
sampling interval
clamping and, 60
defined, 57–58
`Du` and `Dv` functions for determining, 58–59
`du` and `dv` variables for determining, 58
variations in PhotoRealistic RenderMan, 59
sampling rate
aliasing and, 53, 54
clamping and changes in, 60
defined, 52
for low-pass filtering, determining, 57–59
Nyquist frequency and, 53
as sampling interval reciprocal, 57–58
supersampling or oversampling, 54
sampling theorem, 52
Saturn's rings, 485–487
scanline rendering, ray tracing vs., 511
scattering, 531
scattering models
aerial perspective and, 530
color perspective and, 529
Rayleigh scattering, 529, 530, 533, 536, 539
scene description modules (SDMs), 393–394
scene graphs, 315–321. *See also* procedural geometric instancing (PGI)
benefits of, 307
cyclic, 320, 321
defined, 315
instances, 315
iterated function systems, 320
libraries and languages for, 315
limitations of, 307, 321
L-system with additional iteration, 318–320
named node syntax, 315–316
overview, 307
PGI augmentations, 321–330
single-production L-system implementation, 317–318
terminology, 315
tree-structured, 320–321
science
computer graphics vs., 448
iterative method in, 447–448

Team LRN

Science of Fractal Images, The, 429, 433, 438, 489, 575, 579
`scnoise` function. *See* sparse convolution noise
screen space
  mapping for 3D tables for controlling animation, 245
  moving point through solid space, 227, 228, 232–233, 235–243
  object space shading vs., for shading, 103–106
  relating texture space to, 168
SDMs (scene description modules), 393–394
sea surface texture, 140, 141
sedimentary rock strata, 466–467, 468
seed tables for MojoWorld, 597–599
Selene (Moon) texture, 473–477
  highlands/maria, 473
  `luna` procedure, 474–476
  rayed crater, 473–474
self-shadowing. *See also* shadowing
  of clouds, 267
  of gases, 207–208
  of particle systems, 259
  with penumbra, 382–383
  surflets for, 376, 382–383
self-similarity
  defined, 572
  in fractal geometry, 621
  fractal terminology and, 496–497
  in fractals and nature, 572–573
  statistical vs. exact, 435, 575
semblance
  in ontogenetic modeling, 443
  veracity vs., 448
serendipity
  image textures vs. procedural textures and, 15
  procedural techniques and, 2
sexual reproduction, 551, 554
"shade trees" system, 11
`shader` space (RenderMan)
  described, 24
  for solid textures, 25
ShaderLab2 package (Primitive Itch), 130
shading, 7
shading models. *See also specific types*
  anisotropic, 8
  for clouds, 266–267
  defined, 20
  diffuse, 7–8
  for gaseous phenomena, 205, 207
  historical overview, 7–8
  local vs. global, 8
  low-albedo vs. high-albedo, 205
  object space vs. screen space, 103–106
  `plastic` shader, 20–22
  for procedural textures, 20–22
  separation of surface and light properties, 117
  simplifying assumptions in, 7
  specular reflection, 8
shading sample, 15
shading sample point, 15
shadowing. *See also* self-shadowing
  cloud shadowing, 267
  in gases, 207–208
  hardware acceleration for, 208
  in particle systems, 259
  in QAEB tracing, 517
  self-shadowing, 207–208, 259, 267, 376
  volumetric, 207–208
shape
  Boolean characteristic function for, 337–338
  continuous function for, 338
  procedural shape synthesis, 387, 390
`sign` function (RenderMan), `step` function vs., 48
signal processing
  aliasing in, 53–54
  Fourier analysis in, 52–53
  further information, 52
  Nyquist frequency, 53–54
  sampling theorem, 52
SIMD (single instruction, multiple data) computation model, in GPUs, 107–108
`simple_light` shader, 115
sine function
  as band-limited, 57
  gas shape and, 215, 216
  graph of, 32
  overview, 31
  for rotation on helical path, 228
sine wave basis function (MojoWorld), 596
`sinf` function (ANSI C implementation), RenderMan `sin` function vs., 31
single program, multiple data (SPMD) computation model in GPUs, 108
`sintegral` macro, 65
size (area)
  fractal complexity and, 431
  image textures vs. procedural textures and, 14
  photographic texture image problems, 23
  spot size, 166–170
size (storage)
  of image textures vs. procedural textures, 14
  texture size and performance, 110
smoke. *See also* fog; gases; volumetric procedural modeling and texturing

Team LRN

smoke (*continued*)
column, animated, 239–243
column, hypertexture examples, 364–368
column, still image, 219–224
drifting, 365, 366
hardware acceleration for space-filling smoke, 297–298
optimizing, 367–368
rings, 365, 366–367
`Smoke` function, 297–298
`smoke_density_function` for hypertextures, 367–368
`smoke_stream` procedure, 220–224
described, 220
helical path for smoke, 220, 222–224
parameters, 224
`rising_smoke_stream` procedure and, 243
`smoothstep` function (RenderMan)
aliasing and, 57
analytic prefiltering with, 62
C implementation, 30
graph of, 31
overview, 30–31
`step` function vs., 31
`snoise` function
marble texture using, 87–89
perturbed image textures using, 90–91
perturbed regular patterns using, 89–90
soft objects, 3
solid spaces. *See also* animating solid spaces; volumetric procedural modeling and texturing
changing over time, 227, 229–232
development of framework, 209–210
mathematical description of framework, 210–211
moving screen space point through, 227, 228, 232–233, 235–243
overview, 210
three-dimensional tables for controlling animation, 244–254
uses for framework, 210
solid textures. *See also* animating solid spaces; fractal solid textures; real-time procedural solid texturing
for clouds in Maya animation system, 267
color texture, 196
defined, 337
described, 209
fractal, 447–487
fuzzy shapes combined with, 338
as hypertexture precursor, 337–338
overview, 10
real-time procedural solid texturing, 413–427
solid spaces framework for modeling gases, 209–211
solid textured transparency, 233–235
as space-filling functions, 337–338
transparency, 233–235
wood-grain example, 10
space-filling fractals, hypertexture example, 357, 358–359
spaces. *See* coordinate systems; texture spaces; *specific spaces*
sparse convolution basis function (MojoWorld), 593, 595
sparse convolution noise
2D slice of, 74, 82
as fractal basis function, 450, 497
graph of, 73
implementation of, 80–82
power spectrum of, 75, 82
spatial frequency of fractal basis function, 433
spectral synthesis
clamping method for antialiasing, 56, 59–61, 86–87
cloud simulation, 50–51
inverse Fourier transform for, 49
irregular pattern generation using, 85–89
marble synthesis using, 87–89
for modeling gases, 204
noise generation using, 82–83
overview, 48–51
`turbulence` function, 85–87
`specular` function (RenderMan), 21
specular reflection models, 8
`specularcolor` parameter of `plastic` shader, 21, 22
sphere
for hypertexture creation, 353–354
for hypertexture explosion example, 354
implicit formulation of, 270
spherical attractors, 247–249
animating center of attraction, 249
breeze effect using, 247, 248, 252–253, 254
effect of increasing over time, 247
effect over time, 247
extensions of, 249
flow field function, 248–249
geometry of attraction, 249

SPHIGS, 322
spiral paths. *See* helical paths
spiral vortex functions, 249–251
  based on 2D polar coordinate function, 250
  based on conservation of angular momentum, 251
  based on frame number and distance from center, 250–251
  example of effects, 252
  vortex simulation vs., 249
spline functions
  C implementation, 34–35
  as color map or color table, 36
  for colors or points, 35–36
  graph of, 35
  knot values, 34
  overview, 34–37
  reflection texture example, 37
  for tuning procedural textures, 339
  for value noise interpolation, 71–72
SPMD (single program, multiple data) computation model in GPUs, 108
spot noise, 80
spot size
  calculating, 167–170
  "correct" bump mapping using, 171–173
  correcting renderer calculations, 167
  defined, 166
  exaggerating to reduce aliasing, 175
  optimization and verification, 173–175
  renderer calculation of, 166–167
  stretched spots and, 169–170
square waves as fractal basis function, 497
Stanford shading language
  API calls for one face of a cube, 119–120
  API routines, 118–119
  as current sole choice for GPUs, 288
  development of, 131
  early-binding model in, 118
  Heidrich/Banks anisotropic shader, 112–116
  noise-based procedural flame shader, 124–129
  OpenGL API and, 118
  OpenGL lighting model and, 118
  separation of surface and light properties in, 118
  vertex/fragment programming model, 111–112
  volume-rendering shader, 121–124
`star` shader, 47–48
star texture
  feature space for, 46
  geometry of star, 46–47
  image rendered using, 49
  procedural texture generator, 46–48
  random placement pattern using, 92–94
  testing whether point is inside star, 48
Star Trek II: The Wrath of Khan, 257, 258, 578
static bounding volumes, 331
stationary property of `noise` function, 68
statistical self-similarity, exact self-similarity vs., 435, 575
steam. *See also* fog; gases; volumetric procedural modeling and texturing
  breeze effect using attractor, 247, 248, 252–253, 254
  rising from teacup, animated, 236–238
  rising from teacup, still image, 216–219, 220
`steam_moving` procedure
  breeze effect using attractor, 247, 248, 252–253, 254
  for steam rising from teacup, 236–238
`steam_slab1` procedure
  basic slab of steam, 216
  confining steam within cup radius, 216–218
  ramping off gas density for rising steam, 218–219
  `steam_moving` procedure and, 238
  variables, 219
`step` function (RenderMan)
  antialiasing and, 28, 61–62
  box-filtering, 61–62, 64–65
  C implementation, 27
  graph of, 27
  high frequencies generated by, 54, 57
  overview, 27
  for rectangular pulse, 28, 29
  rewriting `if` functions using, 27–28
  `sign` function vs., 48
  `smoothstep` function vs., 31
steps basis function (MojoWorld), 596, 597
stochastic control of gesture, 387, 392–393
stochastic models vs. fractal models, 495–497
stochastic procedures
  for particle creation, 257–258

Team LRN

stochastic procedures (*continued*)
for particle movement, 258
in structured particle systems, 258–259
stochastic sampling
for antialiasing, 55–56, 67
defined, 55
by renderers, 55–56
storage
GPU issues, 103, 289
height fields for terrains, 491, 493–495
surflets for storing hypertextures, 376–384
store-to-memory instruction, GPU lack of, 103
`strata` procedure, 466–467
stratus cloud models, 276, 277–278, 282, 283
stream processors, GPUs as, 110
structured particle systems, 257, 258–259
subsumption architecture, 397
sum table, antialiasing using, 162–163
summed-area table method of antialiasing, 63–64
supersampling
as antialiasing strategy, 54, 66–67, 157
in procedural textures, 66–67
stochastic, 55, 67
by texture instead of renderer, 170–173
surface normal construction for QAEB tracing, 516
surface properties
defined, 116
GPUs and, 117
separating from light properties, 116–117
surface shaders vs. light shaders, 117
surface textures
MojoWorld, 589, 608–609
procedural shape synthesis, 387, 390
volume rendering with, 195–196
surflets, 376–384
advantages over surface-based techniques, 383–384
defined, 376, 378
finding visible surfaces, 379–380
generator for, 381
hierarchical model, 381–382
noise function as, 379
ray tracing in, 379–380, 382–383
selective surface refinement, 380–381
self-shadow with penumbra example, 382–383
shading by, 380
singularities and, 381
steps for generating, 379
surflet model, 377–378
uses for, 376–377, 384
wavelet integration with, 384
as wavelets, 378–379
symmetry, dilation, 431, 573–574
"Synthesis and Rendering of Eroded Fractal Terrains, The," 575–576
synthetic texture models, 11

talking in different moods, 408–409. *See also* texture for facial movement
teleological modeling, 442
temporal antialiasing, verifying, 175
Terragen software, 580
terrains. *See* atmospheric models; earth textures; MojoWorld; procedural fractal terrains
`terran` procedure, 470–472
textural limb animation, 387, 391–395
basic notions, 392
examples, 394, 395
goal determination module (GDM), 394–395
limb motion, 391
overview, 387
related work, 392
scene description modules (SDMs), 393–394
stochastic control of gesture, 387, 392–393
system for, 393–395
textural gesture defined, 393
uses for, 391
texture. *See also* procedural textures (overview)
historical overview, 8–11
L-system specification, 311–312
texture baking, 197–199
texture design methods, 179–201. *See also* user interface for textures
2D mapping methods, 197–199
bump-mapping methods, 183–187
color mappings, 181–182
efficiency issues, 194–195
future of, 199–202
hiding noise artifacts, 179–182
toolbox functions, 179–187
user interface, 187–194
utility textures, 196–197
volume rendering with surface textures, 195–196
texture filtering for real-time procedural solid texturing, 419–421

texture for facial movement, 395–409
addressing an audience, 407
background, 396
basic component movements, 398–399
blinking, 405
bottom-level movement vocabulary, 402–403
example component movement combinations, 399, 400
future work, 409
laughing, 407–408
mixing coherent jitter in component movements, 403, 405
movement layering, 401
movement model, 398–401
noise in layering facial movement, 405–408
opposites of component movements, 399–400
painting with actions, 403–405
problems addressed, 395
related work, 397
searching, 407
small constant head movements, 405–406
talking in different moods, 408–409
Web site, 408
winking, 408
`texture` function (RenderMan)
built-in filtering for, 57
described, 22
texture images, photographic, 22–23
texture mapping
2D methods, 197–199
2D techniques, limitations of, 10
atlas for real-time procedural solid texturing, 416–419
for clouds, 267
evolution of, 10–11
GPU support lacking for vertices, 105
hardware acceleration for smoke and fog, 297–298
photographic texture image problems, 23
in real-time procedural solid texturing, 415–416
shader development and, 130
simple point-sampling approach, 8
texture baking, 197–199
UV mapping, 197–199
texture spaces. *See also specific spaces*
built-in in RenderMan, 24
bump mapping and, 45–46
changing world space to, 168
choosing when defining textures, 24–25
feature spaces, 25
relating to screen space, 168
transforming normal vectors between, 46
user-defined in RenderMan, 24
texturing, 7
three-dimensional tables for controlling animation, 244–254
accessing table entries, 245
breeze effect using attractor, 247, 248, 252–253, 254
combinations of functions, 251–254
common dimensions for, 245
flow into hole in wall, 253–254, 255
functional flow field functions, 246–251
functional flow field tables, 245, 246
nonuniform spacing between entries, 244–245
overview, 244–245
vector field tables, 245
wind effects, 252–254
time vs. space trade-off, 15
time-animated fractal noise, 181
time-dependent textures, 4D noise function for, 84
`tintegral` macro, 65
tinting, solid color texture for, 196
torch hypertexture example, 362, 364
`transform` function (RenderMan), `ntransform` function vs., 46
transparency, solid textured, 233–235
trees
bounding volumes for, 331–332
dilation symmetry in, 573
fractal modeling challenges, 506
L-system for modeling, 309–310
PGI for modeling, 324, 325, 331–332
tree-structured scene graphs, 320–321
tropism
defined, 324
PGI for modeling, 324–326
truth, known vs. proven, 442
turbulence
fBm-valued distortion to emulate, 454–456
fractal character of, 435–436
ontogenetic model of hurricane, 456–458
in QAEB-traced hypertextures, 521
viscous dumping and, 606

Team LRN

turbulence functions
for air current simulation, 238
clamping version, 86–87
in cloud modeling, 270, 271, 281
derivative discontinuities in, 86
`fabs` function with, 369
in flame shader, 129
`fractalsum` function vs., 86
for gases, 211, 214
hardware acceleration for, 291–292
for marble formation from banded rock, 230–232
overview, 368–369
with particle systems, 261
power spectrum of, 85, 86, 87
for rolling fog animation, 238
sine function effects on, 215, 216
for smoke and fog, 297, 298
for smoke stream modeling, 220, 223
for transparency control, 234
for volumetric cloud modeling, 270
turtle graphics symbols in L-system
for bicubic patch modeling, 311
for color and texture specification, 311–312
described, 305
difficulties for human processing, 306
geometric meanings of, 308
labels for debugging or annotation, 312
multicharacter symbols, 311
operating system command execution, 312
parameterized, 310–311
for plant modeling, 308–310, 311
for polygon modeling, 311
query command, 326
for von Koch snowflake modeling, 310–311
type promotion in Stanford shading language, 112

uniform meshed atlas, 416–417
unit interval, remapping, 37–39
upper crossover scale
of fractals, 434
in MojoWorld, 607
user interface for cloud modeling, 278–279
user interface for textures, 187–194
adding parameter range suggestions, 188
color table equalization, 189, 190–192
difficulties in manipulating parameters, 192–193
importance of, 187–188
library of settings, 192, 193
normalizing noise levels, 189–190
parameter proliferation problem, 547–548
polishing the texture, 193
previews (low-resolution), 194
random method for parameter settings, 193
remapping nonintuitive parameter ranges, 188
tracking previous settings, 192
user preset functionality, 193
user-defined functions for raymarcher renderer, 350–352
UV mapping, 197–199

value noise
2D slice of, 74
graph of, 73
interpolation schemes, 71–72
lookup table for, 181
overview, 70–72
power spectrum of, 72, 75
PRN table initialization, 70–71
value-gradient noise, 77–78
Wiener interpolation, 72
value-gradient noise
2D slice of, 74, 78
graph of, 73
overview, 77–78
power spectrum of, 75
Ward's Hermite noise, 73, 74, 75, 78
`vcnoise` function. *See* lattice convolution noise
vector-valued noise (`VecNoise`) function, 450–451
Venus, Coriolis effect in modeling, 458–460
`venus` procedure, 458–460
`vfBm` procedure, 454–455
Virtual Reality Modeling Language (VRML), 333
virtual reality (VR), 581–582, 615
virtualization, CPUs vs. GPUs and, 109
viscous dumping, 606
visibility function in deep shadow map, 208
Vistapro software, 580
`vnoise` function. *See* value noise
volume density functions for modeling gases, 204
volume perturbation for cloud modeling, 299–301
volume rendering
algorithm for gases, 206
QAEB-traced hypertextures, 520–525, 526
ray tracing for gases, 205–206

shader for, 121–124
surface textures for, 195–196
`volume_fog_animation` procedure, 238–239
volumeGas plug-in, 284, 285
volumetric cloud modeling and rendering. *See also* clouds
animating volumetric procedural clouds, 279–283
cirrus cloud models, 275–276, 277, 282, 283
cloud creatures, 273, 278
commercial packages, 284, 285
cumulus cloud models, 272–274, 282
dynamics and physics-based simulations in, 281–282
hardware acceleration for, 298–301
historical overview, 268–269
implicit functions for, 270–272
interactive cloud models, 283–284, 285
jet stream warping, 279–280
microstructure and macrostructure, 270
modeling system, 269–272
particle system for, 269, 282–283
physics-based approach, limitations of, 269–270
rendering, 272–279
simple cloud model, 271
stratus cloud models, 276, 277–278, 282, 283
`turbulence` and `noise` functions in, 270
user specification and control, 278–279
volumetric implicit functions, for cloud modeling, 269, 270–272
volumetric procedural modeling and texturing, 203–224. *See also* gases; volumetric cloud modeling and rendering
a-buffer rendering algorithm, 205
alternative rendering and modeling approaches, 208–209
basic gas shaping, 214–224
constant density medium for, 204
geometry of gases, 211–224
historical overview, 204–205
marble formation animation, 255–256
noise and turbulence functions, 211–214
patchy fog example, 216
power function effects on gas shape, 215–216
ray tracing for volume rendering, 205–206
rendering system, 205–208
sine function effects on gas shape, 215, 216
smoke column example, 219–224
solid spaces procedural framework, 209–211
steam rising from teacup example, 216–219, 220
stochastic functions used for, 203
uses for, 203
volume-rendering algorithm, 206
volumetric shadowing, 207–208
`volumetric_procedural_implicit_function` for simple cloud model, 271
von Koch snowflake
exact self-similarity in, 435
as locally Euclidean, 621
L-system modeling of, 308, 310–311
as simple fractal, 571–572
Voronoi basis function (MojoWorld), 591, 593, 594, 597, 599
vortex functions
based on 2D polar coordinate function, 250
based on conservation of angular momentum, 251
based on frame number and distance from center, 250–251
example of effects, 252
ontogenetic model of hurricane, 456–458
spiral, 249–251
vortex simulation vs., 249
voxel automata algorithm, 359–360
VR (virtual reality), 581–582, 615
VRML (Virtual Reality Modeling Language), 333
`vtransform` function, 36
Vue d'Esprit software, 580

`wallpaper` shader, 92–94
water. *See also* liquid; volumetric procedural modeling and texturing
flow into hole in wall, 253–254, 255
fractal solid textures for, 461–465
noise ripples, 461–463

water (*continued*)
sea surface texture, 140, 141
wind-blown, 463–465
Wavefront's Maya animation system. *See* Maya animation system
wavelets. *See also* surflets
defined, 341, 378
as fractal basis functions, 450, 497
in Perlin's noise function, 340–347
surflet integration with, 384
surflets as, 378–379
weathering, texture for, 197
Web sites
for cellular texturing information, 149
cloud rendering, 284
facial movement examples, 408
genetic programming and genetic art, 557–558
HighEnd3D, 284
Musgrave, F. Kenton, 492, 522, 557, 579, 611
Rooke, Steven, 557
Sims, Karl, 557
for this book, 149, 484, 524
white noise
aliasing and, 68
defined, 67
irregular procedural textures and, 67
physical generation of, 67
repeatable, from PRNs, 67–68
Wiener interpolation for value noise, 72
wind
breeze effect using attractor, 247, 248, 252–253, 254
Coriolis effect, 458–460
jet stream warping for clouds, 279–280
ontogenetic modeling of hurricane, 456–458
turbulence for simulating air currents, 238
wind-blown waters, 463–465
`windywave` procedure, 464–465
winking, 408. *See also* texture for facial movement
World Builder software, 580
World Construction Set software, 580
world space
changing to texture space, 168
changing world coordinate to image coordinate, 168
current space and, 24
described, 24
moving screen space point through solid space and, 228
PGI for object-to-world transformation, 323–324
placement of model into scene and, 315
solid textures and, 25
World Wide Web. *See also* Web sites
procedural geometric modeling and, 333
woven cloth, hypertexture example, 357, 359, 360, 361
`write_noise` function, 212

`xfrog` plant modeling program, 306
z-slicing hypertextures, 35

Team LRN

ABOUT THE AUTHORS

DAVID S. EBERT is an associate professor in the School of Electrical and Computer Engineering at Purdue University. He received his Ph.D. in computer and information science from The Ohio State University in 1991. His research interests are scientific, medical, and information visualization; computer graphics; animation; and procedural techniques. Dr. Ebert performs research in volume rendering, nonphotorealistic visualization, minimally immersive visualization, realistic rendering, modeling natural phenomena, procedural texturing, modeling, animation, and volumetric display software. He has also been very active in the graphics community, teaching courses, presenting papers, chairing the ACM SIGGRAPH '97 Sketches program, co-chairing the IEEE Visualization '98 and '99 Papers program, serving on the ACM SIGGRAPH Executive Committee and serving as editor in chief for *IEEE Transactions on Visualization and Computer Graphics*.

F. KENTON MUSGRAVE, also known as "Doc Mojo," Musgrave is a computer artist and computer graphics researcher with a worldwide reputation. Dr. Musgrave lectures internationally on fractals, computer graphics and the visual arts, and his own computer graphics research. He has developed digital effects for such films as *Titanic* and *Apollo 13*. His images have been widely published and exhibited at international venues, including the Lincoln Center and the Guggenheim Museum in New York City. Dr. Musgrave spent six years in the mathematics department at Yale University working with Benoit Mandelbrot, the inventor of fractal geometry, who credited Musgrave with being "the first true fractal-based artist." He is a founding member of the Algorist school of algorithmic artists and CEO/CTO of Pandromeda, Inc., whose planet-building software product, *MojoWorld,* is the pinnacle of his research. Musgrave has served as director of advanced 3D research at MetaCreations, principal software engineer at Digital Domain, senior scientist at Bethesda Softworks, and assistant professor at George Washington University. Musgrave received his Ph.D. in computer science from Yale University and his M.S. and B.A. in computer science from the University of California at Santa Cruz.

Team LRN

DARWYN PEACHEY is vice president of technology at Pixar Animation Studios in Emeryville, California. He has worked at Pixar since 1988 as a developer of rendering and animation software, as a member of the technical crew on *Toy Story,* and as a technology manager.

Peachey studied at the University of Saskatchewan in Canada, where he received bachelor's and master's degrees in computer science. He later worked as a member of the research staff in the computer science department, where he began his work in computer graphics. Peachey is a member of the Visual Effects Society and the ACM. He has served on several SIGGRAPH and Graphics Interface technical program committees and on the editorial board of the *Journal of Graphics Tools*. His published papers include work in computer graphics and artificial intelligence, and he was one of the recipients of a 1993 Academy Award for the *RenderMan* renderer.

KEN PERLIN is a professor in the computer science department and the director of the Media Research Laboratory and Center for Advanced Technology at New York University. Dr. Perlin's research interests include graphics, animation, and multimedia. In 2002 he received the New York City Mayor's Award for Excellence in Science and Technology and the Sokol Award for Outstanding Science Faculty at NYU. In 1997 he won an Academy Award for Technical Achievement from the Academy of Motion Picture Arts and Sciences for his *noise* and *turbulence* procedural texturing techniques, which are widely used in feature films and television. In 1991 he received a Presidential Young Investigator Award from the National Science Foundation. Dr. Perlin received his Ph.D. in computer science from New York University in 1986, and a B.A. in theoretical mathematics from Harvard University in 1979. He was head of software development at R/GREENBERG Associates in New York, from 1984 to 1987. Prior to that he was the system architect for computer-generated animation at Mathematical Applications Group, Inc. *TRON* was the first movie in which Ken Perlin's name appeared in the credits. He has served on the board of directors of the New York chapter of ACM/SIGGRAPH and currently serves on the board of directors of the New York Software Industry Association.

STEVEN WORLEY has focused his research in computer graphics on appearance and rendering models. His early work on algorithmic textures led to new antialiasing and efficiency adaptations to classical algorithms. In 1996, he introduced the concept of the cellular texturing basis function, which has been widely adopted by most commercial rendering packages. His extensive collaboration with many professional studios has led to the creation of a wide variety of 3D tools. Most recently, his tools for rendering hair and fur have been widely adopted and used in film, TV, and

Team LRN

game development. His company, Worley Laboratories (*www.worley.com*), publishes plug-in tools for various 3D packages.

CONTRIBUTORS

WILLIAM R. MARK was the technical leader of the team at NVIDIA that co-designed the Cg language (with Microsoft) and developed the first release of the NVIDIA Cg compiler. Prior to that, he worked as a research associate at Stanford University, where he co-led the Stanford Real-Time Shading Project with Pat Hanrahan. Starting in January 2003, Bill will join the faculty of the University of Texas at Austin as an assistant professor of computer science. His research interests focus on systems and hardware architectures for real-time computer graphics. Dr. Mark received his Ph.D. from the University of North Carolina at Chapel Hill in 1999.

JOHN C. HART is an associate professor in the computer science department at the University of Illinois at Urbana-Champaign. His research area is procedural methods in computer graphics, including implicit surfaces, texturing, modeling, and animation. He has worked on a variety of procedural modeling and shading projects for Intel, IBM, AT&T, Evans & Sutherland, Kleiser-Walczak, and Blue Sky/VIFX. He received his B.S. in computer science from Aurora University in 1987 and his M.S. (1989) and Ph.D. (1991) in computer science from the Electronic Visualization Laboratory at the University of Illinois at Chicago. He is currently an associate editor for *ACM Transactions on Graphics* and also served on the ACM SIGGRAPH Executive Committee from 1994 to 1999, where he was an executive producer for the feature-length documentary *The Story of Computer Graphics*.

Team LRN